AUTHOR: WIRATUNGA, N.C.

TITLE: Informed selection and use of training examples for knowledge refinement.

YEAR: 2000

THE
ROBERT GORDON
UNIVERSITY
ABERDEEN

Informed Selection and Use of Training Examples for Knowledge Refinement

Nirmalie Chandrika Wiratunga

Thesis submitted to
The Robert Gordon University
for the degree of
Doctor of Philosophy

Abstract

Knowledge refinement tools seek to correct faulty rule-based systems by identifying and repairing faults indicated by training examples that provide evidence of faults. This thesis proposes mechanisms that improve the effectiveness and efficiency of refinement tools by the best *use* and *selection* of training examples.

The refinement task is sufficiently complex that the space of possible refinements demands a heuristic search. Refinement tools typically use hill-climbing search to identify suitable repairs but run the risk of getting caught in local optima. A novel contribution of this thesis is solving the local optima problem by converting the hill-climbing search into a best-first search that can backtrack to previous refinement states. The thesis explores how different backtracking heuristics and training example ordering heuristics affect refinement effectiveness and efficiency.

Refinement tools rely on a representative set of training examples to identify faults and influence repair choices. In real environments it is often difficult to obtain a large set of training examples, since each problem-solving task must be labelled with the expert's solution. Another novel aspect introduced in this thesis is informed selection of examples for knowledge refinement, where suitable examples are selected from a set of unlabelled examples, so that only the subset requires to be labelled. Conversely, if a large set of labelled examples is available, it still makes sense to have mechanisms that can select a representative set of examples beneficial for the refinement task, thereby avoiding unnecessary example processing costs.

Finally, an experimental evaluation of example utilisation and selection strategies on two artificial domains and one real application are presented. Informed backtracking is able to effectively deal with local optima by moving search to more promising areas, while informed ordering of training examples reduces search effort by ensuring that more pressing faults are dealt with early on in the search. Additionally, example selection methods achieve similar refinement accuracy with significantly fewer examples.

Contents

1 Introduction 1

1.1 The Refinement of Knowledge Based Systems 2

1.2 Weaknesses in Existing Systems 3

1.2.1 Hill Climbing Search and the Problem of Local Optima 3

1.2.2 Passive Selection of Training Examples 3

1.3 Project Objectives 4

1.4 The Refinement Tool 5

1.5 Evaluation Domains 5

1.5.1 Student Loans 6

1.5.2 Soybean 7

1.5.3 Manned Maneuvering Unit (MMU) 9

1.6 Synopsis 11

2 Literature Survey 12

2.1 Use of Training Examples for Knowledge Refinement 12

2.1.1 Refinement Driven by an Example At a Time 13

2.1.2 Refinement Driven by Multiple Examples 18

2.1.3 Role of Training Examples for Refinement 20

2.2 Knowledge Refinement as Search 21

2.2.1 Constructive Approach 22

2.2.2 Repair-Based Approach 25

2.2.3 Improving Search Efficiency 26

2.2.4 Dynamic CSPs 27

2.3 Example Selection for Learning Algorithms 28

2.3.1 Uncertainty-Based Classifiers 29
2.3.2 Committee of Uncertainty-Based Classifiers 32
2.4 Conclusion 36

3 Iterative Refinement **38**
3.1 The Iterative KRUSTtool Process 39
3.2 CSP Search Strategies for Knowledge Refinement 42
3.2.1 Knowledge Refinement as Constraint Satisfaction 43
3.2.2 Informed Backtracking with the KRUSTtool 44
3.2.3 Implementation Issues 47
3.3 Adapting the Informed Backtracking Algorithm 47
3.3.1 Latent to Active Examples 48
3.3.2 Prioritising Latent Over Active 49
3.4 Comparison of Backtracking Search for Refinement 50
3.4.1 Experimental Design 50
3.4.2 Results 52
3.5 Exploiting Conflict Knowledge 53
3.6 Conclusion 55

4 Refinement Search Efficiency **58**
4.1 Constrainedness of Refinement Search 58
4.1.1 Evidence From the Recent Refinement Cycle 60
4.1.2 Evidence From How the Problem was Solved 61
4.1.3 Evidence From How the Problem Should be Solved 63
4.2 Ordering Heuristics in Practice 65
4.2.1 Static Ordering 65
4.2.2 Dynamic Ordering 67
4.2.3 Static and Dynamic Combinations 68
4.3 Conclusion 69

5 Informed Selection of Refinement Examples **71**
5.1 The Selective Sampling Process 71
5.2 Selection Guided by Problem-Solving Behaviour 75

5.2.1 Similarity Metric . . . 75
5.2.2 Clustering Technique . . . 77
5.3 Selecting Examples using Clusters . . . 78
5.3.1 Interacting Faults . . . 79
5.3.2 Characteristics of Conflict Pairs . . . 83
5.3.3 Informed Selection Heuristics . . . 84
5.4 Experimental Evaluation . . . 87
5.4.1 Student Loans Domain . . . 88
5.4.2 Soybean Disease Domain . . . 89
5.5 Conclusion . . . 90

6 Informed Selection of Filter Examples **92**
6.1 Filtering Refined KBSs . . . 92
6.2 Cluster-Based Filter Example Selection . . . 95
6.2.1 Simple Selection Heuristics . . . 97
6.2.2 Refinement Extremeness Based Selection Heuristic . . . 98
6.3 Ensemble-Based Selection . . . 101
6.4 Experiments . . . 104
6.4.1 Student Loans Domain . . . 105
6.4.2 Soybean Disease Domain . . . 107
6.5 Conclusion . . . 108

7 Evaluation **110**
7.1 KRUSTtool Variants . . . 111
7.2 Student Loans Domain . . . 113
7.2.1 Error Rate . . . 113
7.2.2 Refinement Cycles . . . 114
7.2.3 Labelling Effort . . . 115
7.3 Soybean Disease Domain . . . 116
7.3.1 Error Rate . . . 116
7.3.2 Refinement Cycles . . . 117
7.3.3 Labelling Effort . . . 118
7.4 MMU Domain . . . 119

7.4.1 Error Rate 120
7.4.2 Refinement Cycles 122
7.4.3 Labelling Effort 123
7.4.4 Computational Overhead 124
7.5 Conclusion 125

8 Conclusion **127**
8.1 Refinement Search 128
8.2 Informed Selection 129
8.3 Experimental Results 131
8.4 Main Contributions 131
8.5 Desirable Extensions 133
8.5.1 Backtracking Search 133
8.5.2 Refinement Example Ordering 134
8.5.3 Example Selection Efficiency 134
8.5.4 Selective Memory Retention 136
8.6 Thesis Summary 137

A Corrupted Student Loans Rule-base in Clips **138**

B Corrupted Soybean Rule-base in Clips **142**
B.1 Soybean Corrupted Rule-base I 142
B.2 Soybean Corrupted Rule-base II 151

C Corrupted MMU Rule-base in Clips **160**

D Interpretation of Results **168**

E Published Papers **170**

List of Figures

1.1 General task of a refinement tool. 2

1.2 A training example from the Student Loans domain. 6

1.3 A training example from the Soybean Disease domain. 8

1.4 A training example from the MMU domain. 10

2.1 The basic operations in KRUST. 13

2.2 The EITHER architecture. 18

2.3 A General Backtracking CSP Algorithm. 23

2.4 Learning with (a) a Negative Bias and (b) a Positive Bias. 30

2.5 Complementing committee based approach with EM. 35

2.6 Filtering Noisy Examples Using a Committee of Classifiers. 36

3.1 The iterative refinement process. 39

3.2 Knowledge refinement as search. 41

3.3 Informed backtracking with the KRUSTtool. 46

3.4 Maintaining refinement information. 47

3.5 Changing behaviour of constraint examples. (a) Refinement Path with a Latent Example. (b) Re-ordering the Latent Example. 48

3.6 Forming training sets that trigger backtracking. 51

3.7 Number of iterations (Basic Algorithms). 52

3.8 Error rate of final refined KBS (Basic Algorithms). 53

3.9 Searching without Conflict-Based re-ordering. 54

3.10 Searching with Conflict-Based re-ordering. 55

3.11 Number of iterations (Conflict-Based re-ordering). 56

3.12 Error rate of final refined KBS (Conflict-Based re-ordering). 56

4.1 Search space (a) without ordering and (b) with ordering 59
4.2 Problem Graph for training examples, A, B and C. 62
4.3 Algorithm combining static and dynamic ordering. 68

5.1 A single iteration of *select-label-refine*. 72
5.2 Sampling within the KRUSTtool. 73
5.3 Clustering and Selecting Examples for Labelling. 74
5.4 Positive Problem Graphs for Example A and Example B. 76
5.5 Clustering of 37 examples from the Soybean domain. 78
5.6 Four rules taken from a corrupted student loans advisor in Clips. 79
5.7 Non-optimal refinement choice triggers backtracking. 80
5.8 Three rules taken from a corrupted student loans advisor in Clips. 81
5.9 Non-optimal refinement choice activates latent example and triggers backtracking. 82
5.10 Over lapping problem graphs for conflict pairs. 84
5.11 Unused examples for student loans domain. 89
5.12 Unused examples for soybean disease domain. 90

6.1 The KRUSTtool filter hierarchy. 93
6.2 The active accuracy filter. 94
6.3 Algorithm for the Cluster-Based Approach. 95
6.4 Analysing Changes in Cluster Content. 96
6.5 Observable usage before and after generalization. 98
6.6 Observable usage before and after specialisation and generalisation. 99
6.7 Proposed KBSs Forming an Ensemble. (a) A Refinement Iteration. (b) Corresponding Ensemble Filter. 102
6.8 The Effects of Filter Example Selection on Error Rate. 106
6.9 The Effects of Filtering on Backtracking. 107
6.10 Number of Iterations for 20 Test Runs. 107

7.1 Relationship between the evaluation strategies. 111
7.2 Error rate for student loans domain. 113
7.3 Number of iterations for student loans domain. 114

7.4 Unused example percentage for student loans domain. 116
7.5 Error rate for soybean disease domain. 117
7.6 Number of iterations for soybean disease domain. 118
7.7 Unused example percentage for soybean disease domain. 119

8.1 Clusters covered by overlapping canopies. 135

List of Tables

2.1 Contingency Table Demonstrating the Use of Training Examples. 21

3.1 Analogy 1: CSPs and Iterative Refinement. 43
3.2 Analogy 2: CSPs and Iterative Refinement. 44

4.1 Constrainedness of training examples using potential refined KBSs. 61
4.2 Refinements and rule activations from the complete problem graph. 64
4.3 Number of iterations for static ordering. 66
4.4 Number of iterations for static + dynamic ordering combinations. 69
4.5 CPU cycles for static + dynamic ordering combinations. 69

6.1 Majority Vote by an Ensemble formed with Proposed Refined KBSs. 103

7.1 Error rate for MMU. 121
7.2 Number of iterations for MMU. 122
7.3 Unused example percentage for MMU. 123
7.4 Comparing computational costs with refinement example selection heuristics CLUSTERREP, K-CLUSTER and CLUSTERMIXED 125

Acknowledgments

I am very grateful for the assistance received from many people. Particularly, thanks go to my supervisor Prof. Susan Craw, who has been a tremendous strength and inspiration throughout my research student life at The Robert Gordon University. I have enjoyed the in-depth discussions we have had over most of the research issues reported in this thesis and (even) enjoyed the challenging questions directed my way. Special thanks to Robin Boswell who has patiently answered all my queries and debugged parts of his core algorithm enabling me to carry on with my experiments. I am grateful to all my colleagues at SCMS, particularly those in C31, with whom I have enjoyed sharing the office. I also wish to thank Alex Wilson for advice on suitable statistical tests, and Dr. Ines Arana for offering to read through my thesis.

I am very grateful to my mother and father, Philo and Brian Wiratunga, who were always supportive of my educational pursuits and put up with the long periods of separation. All this would not be possible if not for the two of them. A heartfelt thank you to my best friend Rosemary Kemp who provided loads of moral support, helped with proof reading and ensured that I did not end up as an all work and no play person. Finally, I hope that you will enjoy reading the research work reported in this thesis as much as I have enjoyed doing it.

Nirmalie Wiratunga

Chapter 1

Introduction

Decision support systems such as medical diagnosis, advisory systems, and design systems typically require extensive knowledge of the subject at hand. Systems that aim to capture and model the underlying knowledge from an expert in the particular application area (domain) are referred to as *Knowledge Based Systems* (KBSs). The KBS development life cycle consists of knowledge elicitation, knowledge representation, debugging and maintenance. Successful completion of this cycle is heavily reliant on the interaction between the developer and the expert. Therefore, tools that can help automate some or most of the tasks involved during this cycle are certain to reduce the expert's and developer's effort. *Refinement tools* focus on the debugging and maintenance stages, and they seek to automate these two stages by identifying and correcting mismatches between the world modeled by the KBS and the real world. Episodes of expert problem solving represent the real world and are maintained as *training examples*. Crucial to the successful operation of the refinement tool is the availability of a set of training examples representative of the expertise captured by the KBS.

The purpose of this chapter is to introduce knowledge refinement, project motivations and an overview of future chapters. The task of knowledge refinement and a general formalism is outlined in Section 1.1. Weaknesses in existing refinement tools in Section 1.2 forms project motivations and objectives in Section 1.3. Section 1.4 introduces the KRUSTWorks project, since the research reported in this thesis was carried out as part of that project. The structure of training examples with respect to three application domains and a synopsis of the thesis follows in Sections 1.5 and 1.6.

1.1 The Refinement of Knowledge Based Systems

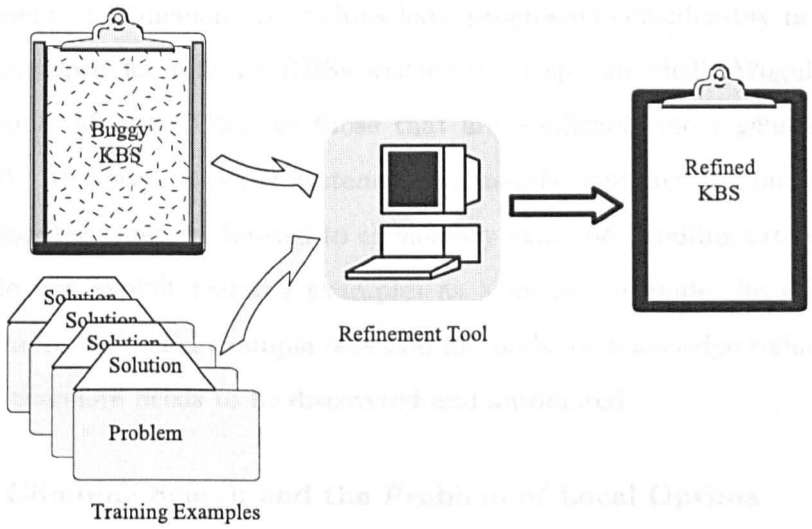


Figure 1.1: General task of a refinement tool.

The task of refinement tools can be viewed as learning to adapt knowledge (Figure 1.1). The input to the refinement tool is the buggy KBS and a set of training examples each comprising the expert's solution given a problem. The output is the refined KBS. Here, we are particularly interested in the refinement of rule-based systems, and formalise the task of KBS refinement (knowledge refinement) as follows. Given a KBS, that does not completely reproduce an expert's problem solving behaviour for a set of training examples, the task of the refinement tool is to:

1. discover faulty problem solving behaviour of the KBS, by identifying mismatches between the expert's solution and the KBS's solution for a given problem;
2. propose one or more potential refinements to rules in the KBS modifying its problem solving behaviour such that the expert's solution can be achieved;
3. implement the refinements as refined KBSs; and
4. select the *best* refined KBS if many are proposed, with the aim of improving the KBS's empirical performance on the training examples.

Steps 2 and 3 involve the *refinement algorithm.* The emphasis of Steps 1 and 4 is on the use and selection of training examples. Accordingly, training examples provide fault evidence in Step 1, and provide a test-bed for evaluating refined KBSs in Step 4.

1.2 Weaknesses in Existing Systems

The development of refinement algorithms have progressed considerably in recent years, from tools developed for refining KBSs written for a specific shell (Wogulis & Pazzani 1993, Ourston & Mooney 1994) to those that are applicable more generally (Craw & Boswell 1999). However, existing systems have mainly concentrated on improving the refinement algorithm and are limited to elementary example handling procedures. These procedures do not exploit training examples as a means to guide the refinement process. Furthermore, informed example selection methods for knowledge refinement is non-existent, and therefore needs to be discovered and automated.

1.2.1 Hill Climbing Search and the Problem of Local Optima

Typically refinement tools adopt an incremental approach where each application of the refinement algorithm attempts to fix one or more, but typically not all of the training examples that provide fault evidence. The refinement task is sufficiently complex that the space of possible refinements demands a heuristic search. EITHER (Ourston & Mooney 1994) and FORTE (Richards & Mooney 1995) try to fix the outstanding fault that is indicated by the *largest* number of examples, and selects the refinement with the *fewest* changes to rules which are *farthest* from the outcome. Craw's (1996) approach to refinement also applies hill-climbing search where the refined KBS that fixes the largest number of examples is selected, but generates many refined KBSs designed to fix a single incorrectly solved training example at a time. The result is that refinement tools are dogged by the standard hill-climbing problem of getting caught in local maxima, where the performance of the KBS must be reduced before an overall improvement can be gained. However, most refinement tools do not explicitly handle this problem, instead they employ induction or non-generic fixes that circumvent the situation.

1.2.2 Passive Selection of Training Examples

Refinement tools have achieved the initial goal of reducing expert and developer effort by adopting abductive, deductive and inductive techniques that automate the knowledge debugging and maintenance stages in the KBS development cycle (Mooney 1997). The success of these techniques depends on the availability of a representative set of training

examples. Typically, access to such a set is only possible through considerable interaction with the expert. Presently, all refinement tools adopt a passive approach to training example selection. This means that refinement tools do not actively select training examples that are desirable from a refinement point of view, instead they expect a training set that covers the KBS's expertise to be available. Clearly, this is not a realistic expectation, where a busy expert cannot be expected to provide solutions to hundreds of random problem situations with the hope of covering the gamut of expertise. Even if a comprehensive training set is available refinement tools must still be able to deal with skewed training example distributions, because refinement tools aim to improve the empirical performance of the KBS when employing training examples as a test-bed during refinement selection.

1.3 Project Objectives

The weaknesses of refinement tools just identified provides new directions and opportunities for further research within the knowledge refinement context. The objectives of this research are two fold:

- *Refinement search*, to improve the effectiveness of incremental refinement by solving the local maxima problem. The approach adopted with respect to this problem is to modify the sequence of incremental refinements by retreating to previous states of refinement. Additionally, a more pro-active approach to incremental refinement suggests an investigation into refinement sequence pre-planning, with the aim of improving the efficiency of incremental refinement.

- *Example selection*, to enable refinement tools to actively select training examples given the refinement purpose of providing fault evidence or forming a test-bed upon which proposed refinements can be competitively evaluated.

Interestingly the strategies to achieve these objectives necessitated cross-fertilisation between knowledge refinement, search methods from the constraint satisfaction paradigm and selective sampling techniques from the machine learning community.

1.4 The Refinement Tool

The research reported in this thesis was carried out as part of the larger KRUSTWorks project[1], which aims to develop a generic knowledge refinement framework. Given a specific rule-base shell, this framework is used to generate a refinement tool, a KRUSTtool, by re-using core refinement modules. These modules are applied to generic knowledge structures which model the problem solving behaviour of the rule-base. The structures are formed by translators that work on the specific rules and the associated traces (Craw & Boswell 1999). The currently developed framework is able to deal with faulty KBSs implemented in shells incorporating reasoning strategies that can be forward-chaining, backward-chaining or both. We will work with a KRUSTtool, in particular the Clips KRUSTtool. However, developed methods with regards to the utilisation and selection of training examples need not be specific to just this KRUSTtool and should be applicable in a wider context.

1.5 Evaluation Domains

The evaluation is based on two artificial domains and one real domain. Ideally, with each test domain we would have access to a buggy KBS and a sufficiently large data set for training and testing purposes. Unfortunately, access to an industrial expert system during its debugging stage is hard to achieve. Instead we can obtain a KBS assumed to be correct, and create a faulty KBS by adding controlled corruptions to a copy of the original. The advantage of this approach is that in situations where training examples are not readily available, or are not sufficiently representative of the KBS's expertise, the original KBS can be exploited to generate new examples. We ensure that corruptions to the KBS are refinable by the KRUSTtool, because here we are interested in improving example selection and utilisation methods, in contrast to improving refinement operator diversity. Therefore, the types of corruptions are restricted to four KRUSTtool refinement operators:

Delete Rule : triggered by an extra rule corruption;

Remove Condition : triggered by an extra condition in a rule or an extra disjunction in a rule condition;

[1]The KRUSTWorks project is supported by EPSRC grant GR/L38387 awarded to Susan Craw.

Change Comparison Value : triggered by a rule condition with an incorrect comparison value; and

Change Comparison Operator : triggered by a rule condition with an incorrect mathematical comparison operator.

Essentially, introducing corruptions that the refinement tool is unable to fix will not help ascertain anything about example selection and utilisation. What we need is *pro-active* corruptions designed to provoke fault evidence.

Typically, training examples must be converted in to a programming environment specific format, where the environment is the one in which the KBS was developed. The problem part of the training example is maintained as a set of observables represented as facts and is employed to initialise the KBS with the problem task. The solution part is represented in a format enabling easy comparison with the KBS's solution.

1.5.1 Student Loans

The Student Loans Advisor has been widely used to evaluate various knowledge refinement systems (Murphy & Pazzani 1994, Pazzani & Brunk 1991). The purpose of the advisor is to determine whether a student given his/her circumstances should or should not repay a US educational loan. The data set and the KBS in Prolog can be obtained from the UCI repository (Blake, Keogh & Merz 1998). The data set consists of 1000 labelled examples. We use a Clips version of the student loans KBS containing 20 rules. For experimentation purposes the KBS was corrupted by introducing 5 corruptions (see Appendix A): an extra rule, a changed comparison operator and an extra condition in 3 rules.

```
Observables:                (male student44)
                            (absence student44 9)
                            (enrolled student44 uci 1)
                            (unemployed student44 no)
                            (disabled student44 no)
                            (enlisted student44 no)
Expert Solution (label):    (no_payment_due student44 yes)
```

Figure 1.2: A training example from the Student Loans domain.

Figure 1.2 shows a typical training example from this problem domain, comprising the

observables and the expert's solution (also referred to as the example's label). This training example is a positive instance of the class **no_payment_due**, and corresponds to student, **student44**. In contrast a negative instance of class **no_payment_due**, would consist of a label such as (**no_payment_due studentX no**). A domain comprising of just positive or negative instances of a single concept is referred to as a binary classed domain. Of the 1000 examples 643 are positive and the rest are negative.

A set of 100 training examples and a disjoint set of 100 testing examples are randomly selected from the 1000 data set for experimentation. The Clips KRUSTtool is run with increasing subsets of the 100 training examples and the refined KBS is evaluated on the independent test examples, with final results typically averaged over ten runs. The majority of experiments reported in the thesis using the student loans adviser is designed in this manner unless otherwise stated.

1.5.2 Soybean

The Soybean disease diagnosis system has been widely used to evaluate various machine learning algorithms and refinement systems (Carbonara & Sleeman 1999, Schlimmer 1988, Michalski & Chilausky 1980). Given several symptoms of disease in soybean plants this system is able to classify them into one of 19 possible diseases. Unlike the binary class Student Loans domain, here we have a multi class domain. A large data set consisting of 307 examples, and a small data set consisting of 47 examples was obtained from the UCI repository (Blake et al. 1998). A data set of 337 labelled examples for experimentation was formed by merging the large and small soybean data sets and selecting those examples classified in 15 of the 19 classes. While each of the 15 classes on average had 20 representative examples the remaining 4 classes seemed unjustified because they were represented by just 17 examples in total.

The original Soybean KBS has certainty-factors associating probabilistic weights to certain disease symptoms (Michalski & Chilausky 1980). As the KRUSTtool presently does not have certainty-factor related refinement operators this original Soybean KBS was not suited for our task. Carbonara & Sleeman (1999) dealt with this problem by translating the rules into a non-probabilistic form by deleting any symptoms from the theory that had a weight less than 0.8. The translated KBS has only a 12.3% accuracy on the labelled examples and is ideal for evaluation of a refinement tool's refinement operator capabilities.

However, for evaluation of example use and selection techniques it makes sense to work with a training set that has a more balanced number of true and false examples. Therefore, a new Clips soybean KBS with 46 rules was created by incorporating rule chaining into the rule set generated by `c4.5rules` (Quinlan 1993). Two corrupted versions of this KBS are used, with the aim of achieving two different levels of corruption:

- corrupted in 7 places, by adding and modifying antecedents in rules covering 4 of the 15 classes (see Appendix B.1); and
- corrupted in 13 places, by adding and modifying antecedents in rules covering 8 of the 15 classes (see Appendix B.2)

These two KBSs tend to have a flat structure when compared to the more straggling Student Loans KBS. The reason for this is that the initial Soybean KBS was generated using `c4.5rules` and therefore, inherits the flat structure characteristic of induced rules.

```
Observables:                (fruit_pods plant1 norm)
                            (fruit_spots plant1 absent)
                            (fruiting_bodies plant1 present)
                            (leafspots_halo plant1 absent)
                            (leaves plant1 abnorm)
                            (plant_growth plant1 abnorm)
                            (plant_stand plant1 norm)
                            (seed plant1 norm)
                            (roots plant1 norm)
                            (stem plant1 abnorm)
                            (mycelium plant1 absent)
                            ⋮
Expert Solution (label):    (diagnosis plant1 diaporthe-stem-canker)
```

Figure 1.3: A training example from the Soybean Disease domain.

Figure 1.3 shows a typical training example from the Soybean problem domain, comprising the observables and the expert's solution. Here, a training example is described using 35 observables and is clearly more realistic than a training example from the Student Loans domain. A set of 100 training examples and a disjoint set of 100 testing examples are randomly selected from the 337 data set. The Clips KRUSTtool is run with increasing subsets of the 100 training examples and evaluated on the 100 testing examples, with final results typically averaged over ten test runs.

1.5.3 Manned Maneuvering Unit (MMU)

The MMU fault diagnosis system is a real application consisting of 104 rules with each rule on average comprising 8 antecedents. The system is written in Clips and was used in (Boswell & Craw 2000) to evaluate the KRUSTtool's refinement operators. NASA's MMU is a one-man, nitrogen-propelled backpack that attaches on to an astronaut's spacesuit. This jet pack enables the astronaut to fly untethered in or around the orbiter. Given information about the MMU's operator controls and measurements the MMU system provides automatic fault diagnosis, and generates recovery procedures for the MMU. A data set of 100 labelled examples were formed by augmenting the 6 examples supplied with the MMU system, with 94 manually generated examples. Manual generation aimed at covering the problem space uniformly and involved determining the range of possible values for observables by examining the existing examples and the rule-base. The original KBS was then used to determine the *correct* diagnosis for the 94 examples. Essentially the original KBS acted as the expert or oracle providing labels for the 94 examples.

The expert reasoning modeled by the MMU KBS is complex and difficult to monitor when compared to the Student Loans or the Soybean KBSs. The primary contributory factor to this complexity, is the non-monotonic behaviour of the MMU KBS, whereby facts asserted by rules are retracted in subsequent rule activations. Additionally, the presence of negated conditions, antecedents comprising both disjunctions and conjunctions, and different rule priorities are further contributory factors.

The original KBS had two corruptions: a generalised disjunction in one rule and an extra negated condition in another rule. For experimentation purposes a copy of the original faulty KBS was further corrupted by introducing 12 corruptions to 12 rules (see Appendix C): a generalised disjunction in 2 rules, an extra condition in 8 rules, and an extra negated condition in 2 rules. Note that the corruptions were random and not controlled, because it is difficult to anticipate the inherent behaviour of the MMU KBS. Refining a faulty MMU KBS requires significant processing power and refinement time. Therefore, for development purposes Student Loans and Soybean domains have the added advantage of being economical, simpler and easier to monitor. However, the MMU problem domain being a real one, makes it an ideal candidate for testing of implemented methods in Chapter 7.

```
Observables:                (side a on)
                            (side b on)
                            (aah off)
                            (gyro off)
                            (fuel-used-a 0)
                            (fuel-used-b 0)
                            (xfeed-a closed)
                            (xfeed-b closed)
                            (tank-pressure-was a 500)
                            (task-pressure-was b 500)
                            (tank-pressure-current a 500)
                            (tank-pressure-current b 500)
                            ⋮
Expert Solution (label):    (conclusion cea failure side-a)
```

Figure 1.4: A training example from the MMU domain.

Figure 1.4 shows a typical training example comprising the observables and the expert's solution. The number of observables describing each example is not fixed, but on average an example is described by 40 observables. The incremental experimental design using 100 training and 100 testing examples is not suitable, because the MMU data set is comparatively smaller than the sets used for Student Loans and Soybean. Therefore, for this domain we adopt the 5x2 cross-validation (5x2cv) experimental design method proposed in (Dietterich 1998). This involves 10 test runs obtained as follows:

Repeat 5 times

Randomly partition the data set into 2 equal-sized sets S_1 and S_2

train on S_1 and test on S_2

train on S_2 and test on S_1

The 5x2cv design is well suited here, because partitioning the 100 data set gives us a sufficiently large training and testing set of size 50. Furthermore, 5x2cv comes recommended as it has good type I error, i.e. low probability of incorrectly detecting a difference when no difference exists.

1.6 Synopsis

The first part of the thesis concerns the project background. Chapter 1 has provided a general overview of knowledge refinement, objectives and problem domains. The literature survey in Chapter 2 is organised under three sections: knowledge refinement; constraint satisfaction search strategies; and selective sampling.

The second part of the thesis deals with refinement search. Incremental refinement with the KRUSTtool is described in Chapter 3. Here, the local optima problem of hill-climbing search is highlighted and strategies to deal with this problem are incorporated with the KRUSTtool. Evaluation of search strategies on the student loans domain indicate that solving the local optima problem improves the KRUSTtool's effectiveness but efficiency can be undermined. Therefore, in Chapter 4 we investigate how KRUSTtool's efficiency might be improved, and for this purpose training example ordering strategies that effect the sequence of incremental refinements is presented.

The third part of the thesis concentrates on informed selection of training examples. In Chapter 5 a clustering framework which aids the KRUSTtool to actively select training examples for providing fault evidence is described. The emphasis of Chapter 6 is active selection of training examples for refinement filtering. One of the selection techniques builds on the clustering framework in Chapter 5 while the others exploit the diversity amongst generated refined KBSs when voting for or against selecting examples for refinement filtering. We use student loans and soybean domains to evaluate several selection strategies.

The rest of the thesis consists of evaluations, conclusions and future directions. Chapter 7 reports on experiments carried out on all three problem domains, comparing several KRUSTtool variants formed by combining refinement search and example selection methods. Finally, Chapter 8 presents project conclusions and suggestions for future research.

Chapter 2

Literature Survey

The survey on refinement systems places particular emphasis on the use of training examples. The refinement process is driven by training examples and involves a search for the best refinement through the space of possible refinements. This highlights the need for effective search strategies that are to some extent lacking in current refinement systems. For this purpose, a significant part of this chapter is also dedicated to the study of efficient search algorithms employed by a different class of AI problems. Another issue that is not sufficiently addressed by refinement systems, is example selection mechanisms. This necessitated an investigation of example sampling strategies that are currently employed by the machine learning community. Therefore, this literature survey consists of three distinct sections:

- the use of training examples for knowledge refinement (Section 2.1);
- search strategies that might be beneficial when searching the space of possible refinements (Section 2.2); and
- training example selection techniques that are employed by machine learning algorithms (Section 2.3).

2.1 Use of Training Examples for Knowledge Refinement

Training examples can be processed a single example at a time, where the refinement system reacts to fault evidence by a single training example. Alternatively, multiple train-

ing examples might be processed as a batch, where the refinement system implements refinements once fault evidence provided by all training examples is analysed.

2.1.1 Refinement Driven by an Example At a Time

The KRUST (Craw 1996) knowledge refinement system iteratively refines a faulty KBS by processing a single example at a time and can be applied to KBSs from both classification (Craw & Hutton 1995) and design domains (Boswell, Craw & Rowe 1997). Figure 2.1 illustrates the knowledge refinement tasks undertaken by KRUST. The input to KRUST consists of the faulty KBS and a set of training examples, $e_1, \ldots, e_n$. At each iteration of the refinement algorithm, a single training example referred to as the *refinement example*, is presented to KRUST. If the refinement example is correctly solved then refinement is not required, otherwise the *fault evidence* is employed to identify the cause of the faulty problem solving behaviour, generate several potential refinements and implement them as refined KBSs. The generation of multiple refined KBSs is a unique feature of KRUST, and at the filtering stage less promising refined KBSs get discarded.

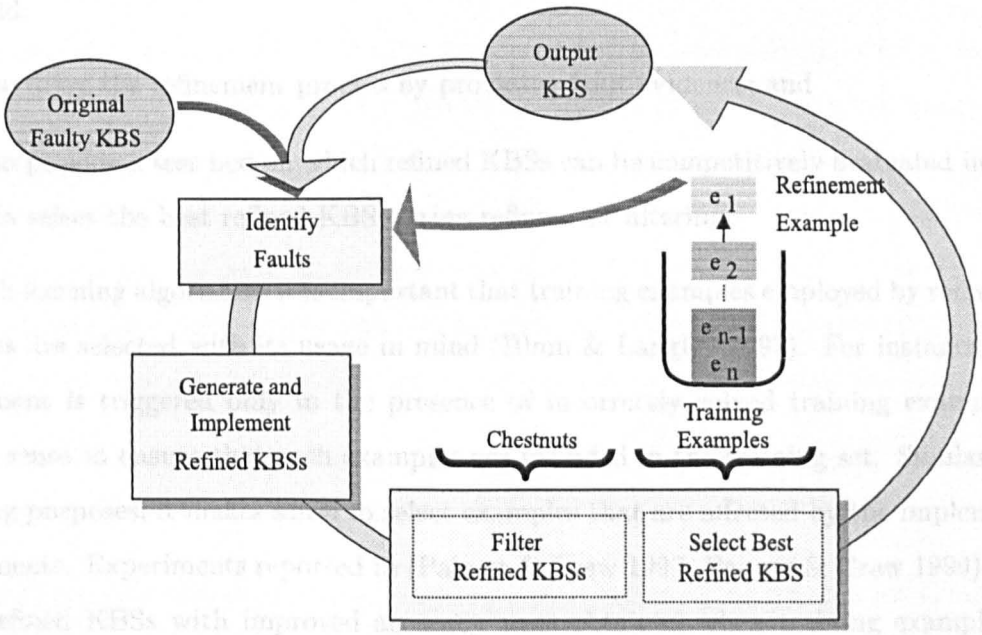


Figure 2.1: The basic operations in KRUST.

Several filters are employed by KRUST in order to select the best refined KBS. KRUST prefers refined KBSs with highest accuracy on previously solved examples. Problems can arise when proposed refined KBSs have the effect of undoing previous refinements,

thereby causing inconsistencies with previously solved refinement examples. KRUST deals with these inconsistencies by adding a new rule that explicitly fixes the fault exposed by the current refinement example. Although such a strategy ensures that the current refinement example is solved without undoing previous refinements, the outcome is a specialised refinement that overfits the example. However, if refined KBSs consistent with previously solved examples are found, the *accuracy filter* selects the refined KBS with highest accuracy on the remaining training examples. KRUST can also incorporate a *chestnut filter*, which ranks refined KBSs by accuracy on a selected subset of special examples called *chestnuts*. An expert identifies these chestnuts as compulsory problem solving tasks that the selected refined KBS must solve correctly. Such a filter can only be employed if chestnuts are provided by an expert prior to the refinement process. The outcome of the filtering stage is the *best* refined KBS which is the output KBS in this iteration, and the input KBS in the subsequent iteration. Generally the filters enforce a greedy hill-climbing search that runs the risk of getting caught in local optima.

The operation of KRUST suggests the role of training examples for refinement to be two-fold:

- to drive the refinement process by providing fault evidence; and
- to provide a test bed on which refined KBSs can be competitively evaluated in order to select the best refined KBS during refinement filtering.

As with learning algorithms it is important that training examples employed by refinement systems are selected with its usage in mind (Blum & Langley 1997). For instance, since refinement is triggered only in the presence of incorrectly solved training examples, it makes sense to ensure that such examples are included in the training set. Similarly, for filtering purposes, it makes sense to select examples that are affected by the implemented refinements. Experiments reported in (Palmer & Craw 1997, Palmer & Craw 1996), show that refined KBSs with improved accuracy were obtained when training examples are carefully selected to have at least one example that provides fault evidence, compared to random selection. They also introduce the notion of selecting *awkward examples* for filtering purposes, to ensure that refined KBSs are evaluated on examples relevant to the refined KBSs being evaluated. In Figure 2.1, adding the awkward cases filter will result in an extra filter level after the chestnut filter but before the selection of the refined KBS with

highest accuracy. Essentially, such a filter aims to identify examples that are affected by the refinement by selecting examples that meet certain criteria. These criteria are derived according to the effect of the proposed refinement on rule activations and on the system solution.

- *Results changed criteria*, select examples that as a result of refinement are now solved differently.
- *Paths diverge criteria*, select examples that in addition to results changed criteria, are now incorrectly solved as a result of over-specialisation.
- *Maximally false criteria*, select examples that in addition to results changed criteria, are now incorrectly solved as a result of over-generalisation.

Identifying examples that fall into these criteria involve examining changes in system solution, detailed analysis of changes in rule activations and fact assertions. For KBSs with large numbers of rules, fact assertions and retractions a detailed analysis with each example is impractical.

The COAST (Rajamoney 1990) refinement system processes a single training example at a time and incorporates a filtering mechanism that aims to maintain consistency with previously solved examples. COAST refines KBSs that model knowledge about physical process theories. The refinement process is triggered when the predicted behaviour of a physical process scenario, is different from the observed behaviour as captured by a training example. Once refined theories are proposed, the best is selected by means of several filters. The filtering mechanism, *exemplar-based theory rejection* ensures that:

- refinements are accepted only when they are consistent with observed behaviours of previously solved examples; and
- the best refinement is selected by evaluation on a selected subset of relevant examples that are affected by the refinement.

The selected refinement is incorporated into the KBS. Close parallels can be drawn between the motivations behind KRUST's awkward example filter, and COAST's filter mechanism based on examples affected by the implemented refinement. Unlike KRUST, COAST must generate refinements that are consistent with previously solved examples. However, it is

not clear from the literature whether COAST is always able to generate consistent refinements, and if not, what remedial procedures might be required. An interesting feature of COAST is the manner by which training examples are indexed according to sub-parts of the theory that they exercise. This enables quick identification of affected examples by changes to parts of the theory during filtering.

The explanation based approach employed by ODYSSEUS (Wilkins 1988, Wilkins 1990) extends incomplete domain theories developed using the MINERVA expert system shell. ODYSSEUS observes an expert operating in a diagnosis problem domain, and monitors the explanation generated by the system for the expert's actions. Each action is analysed individually, analogous to processing a single example at a time. Unlike training examples used by most other refinement systems, here each training example constitutes a single feature value instead of a set of feature value pairs. The resulting training example is far more fine-grained than the typical training example employed by other refinement systems. Furthermore, as many training examples form a single diagnostic session of an expert, the order of examples is important as it captures the implicit information about the current reasoning and possible diagnosis. Refinement is triggered when ODYSSEUS is unable to explain an expert's action. Once a refinement is generated it needs to be validated. For this purpose ODYSSEUS has associated with each type of refinement a validation procedure, called a *confirmation decision procedure* (CDP). For instance in a medical domain if a refinement suggests the addition of a new *clarifying question* to the patient, the associated CDP will check that the clarifying question is linked to many disease hypotheses and can effectively eliminate a high percentage of these hypotheses. Similarly if a refinement involves the addition of a rule, the associated CDP will check whether the rule meets the *goodness* measures, such as simplicity, redundancy etc. It is clear that ODYSSEUS's approach to refinement filtering exploits the underlying domain theory and structural information of the proposed refinement. This is in contrast to selecting refinements based on accuracy on training examples.

The FOCL (Pazzani & Kibler 1990) system learns relational concepts from an existing rule base using explanation based learning and inductive learning. KR-FOCL (Pazzani & Brunk 1991), is a refinement system that complements the explanation based learning component of FOCL. Learning experiences are recorded by FOCL and contain information about rules that were used for learning, any conditions or rules that needed to be induced,

together with details of correctly and incorrectly solved examples. KR-FOCL uses these experiences to identify potential faults in the rule base by applying several refinement heuristics. Like KRUST, refinement generation in KR-FOCL is triggered by a single example. However, unlike the explicit fault evidence that triggers KRUST's refinement process, with KR-FOCL, fault evidence is established by reasoning from FOCL's learning experience. Consequently, several refinements are proposed, and user-interaction is exploited to select the best one. Therefore, like ODYSSEUS, training examples are used only to detect faults and not for filtering purposes.

CLIPS-R is a refinement system that is built explicitly with the refinement of Clips KBSs in mind (Murphy & Pazzani 1994). Interestingly, CLIPS-R executes all examples on the rule-base to establish the sequence in which examples are to drive the refinement process. Essentially, CLIPS-R forces a sequence on refinements by dealing with examples that seem to indicate the most pressing faults in the rule base first. For this purpose a tree structure is constructed where each node represents a rule activation and tree traversal captures the sequence of rule activations leading to a leaf node. The leaf nodes contain groups of examples with similar rule activations. It is possible that an example can be in one or more leaf nodes depending on rule activations. Additionally, each node records the proportion of examples incorrectly solved with respect to the rule activation represented by that node. This means that the root will have the highest error-rate, while leaf nodes will typically have lower error-rates as examples get dispersed to various branches with tree traversal. CLIPS-R selects the examples at the leaf node with highest error-rate to drive the refinement process. Enforcing an order on training examples in this manner will affect the sequence in which refinements get implemented. However, ordering based on error rate alone can be adversely affected when example distribution is skewed. Nevertheless, the idea of processing examples in a predetermined sequence is an interesting concept that we have not seen with other refinement systems discussed so far. Note however, that although CLIPS-R uses all training examples to form the tree, once the relevant leaf node is identified, it processes a single example at a time. Like KRUST, CLIPS-R generates several refinements and evaluates them on all training examples, selecting a single refinement with highest accuracy. Hill climbing in this manner makes CLIPS-R susceptible to getting caught in local optima just like KRUST.

2.1.2 Refinement Driven by Multiple Examples

Refinement systems that deal with a batch of training examples, must process fault evidence provided by all training examples in the batch before generating refinements. Dealing with multiple fault evidence entails establishing suitable criteria that helps prioritise repairs. The approach adopted by EITHER and FORTE, is a greedy algorithm that implements those refinements that fix the highest number of incorrectly solved examples. Both refinement systems have been developed primarily for problems in the classification domain.

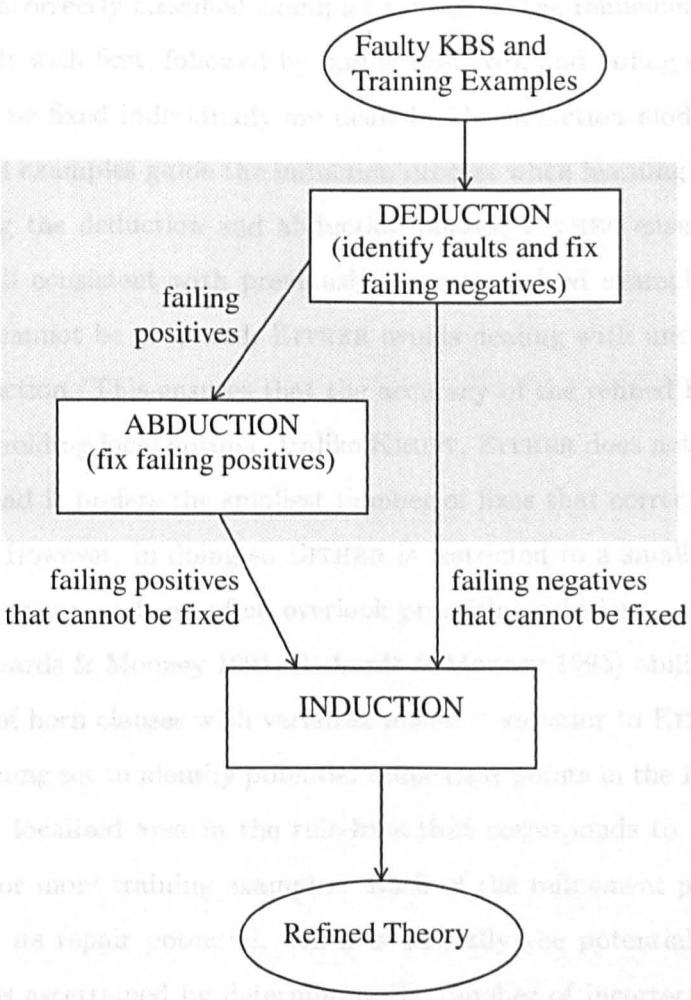


Figure 2.2: The EITHER architecture.

EITHER (Ourston & Mooney 1994, Mooney & Ourston 1991) is a knowledge refinement system that can fix faulty Prolog KBSs restricted to horn clauses without variables. EITHER operates in batch mode, processing a complete set of training examples at once. The

set of training examples consists of correctly and incorrectly classified examples. Correctly classified examples help focus the refinement while incorrectly classified examples provide fault evidence and drives refinement. EITHER groups the incorrectly classified examples into two categories based on the type of fault evidence they provide:

- failing positives, examples not provable in their own class; and
- failing negatives, examples provable in classes other than their own class.

The refinement algorithm consists of three distinct modules (see Figure 2.2) and enforces an order on the incorrectly classified examples that drive the refinement process. Failing negatives are dealt with first, followed by failing positives, and failing examples of either type that cannot be fixed individually are dealt by the induction module. Additionally, correctly classified examples guide the induction process when learning new rules and antecedents. During the deduction and abduction phases, EITHER ensures that proposed refinements are all consistent with previously correctly solved examples. When consistent refinements cannot be proposed, EITHER avoids dealing with uncorrected examples by initiating induction. This ensures that the accuracy of the refined KBS is always improved, thereby avoiding local optima. Unlike KRUST, EITHER does not generate multiple refinements, instead it prefers the smallest number of fixes that correct the largest number of examples. However, in doing so EITHER is restricted to a small proportion of the refinement search space, and can often overlook promising solutions.

FORTE's (Richards & Mooney 1991, Richards & Mooney 1995) ability to refine Prolog KBSs consisting of horn clauses with variables makes it superior to EITHER. FORTE uses the complete training set to identify potential refinement points in the KBS. A refinement point is simply a localised area in the rule-base that corresponds to the fault evidence provided by one or more training examples. Each of the refinement points are assigned a score reflecting its repair potential, which is basically the potential increase in accuracy. The score is ascertained by determining the number of incorrectly solved training examples that will be correctly solved as a result of fixing the identified refinement point. For instance, if a potential refinement for a given refinement point would result in five incorrectly solved training examples being solved correctly, then the refinement point is assigned a score of five. Notice that the refinement score does not reflect the number of correctly solved training examples that may get unsolved as a result of the proposed

refinement. Once scores are assigned, the refinement points are sorted by highest to lowest refinement potential. Essentially, this scoring and sorting mechanism helps FORTE to prioritise refinements, however like the tree structure employed by CLIPS-R, sorting can be adversely affected when example distribution is skewed.

FORTE generates refinements for each identified refinement point. The generated refinements are scored by the actual increase in accuracy on training examples. FORTE stops generating refinements when the potential of the next refinement point is less than the actual accuracy increase of the best refinement so far. FORTE's global view of refinement generation based on accuracy on all training examples is once again an attempt to avoid getting caught in local optima, however, this is not guaranteed.

The AUDREYII (Wogulis & Pazzani 1993) refinement system can be applied to KBSs that deal with binary classification problems. AUDREYII adopts a two-stage process:

- a specialisation stage that deals with *false negatives*, which are positive examples incorrectly classified as negative; and
- a generalisation stage that fixes *false positives*, which are negative examples incorrectly classified as positive.

During specialisation, AUDREYII's hill-climbing search selects the refinement that fixes the most number of false negatives. In the generalisation stage, false positive examples are randomly selected and processed one at a time. This two-stage approach can be viewed as a form of example ordering, and is distantly related to EITHER's deduction module dealing with failing negatives, followed by the abduction module dealing with failing positives.

2.1.3 Role of Training Examples for Refinement

Table 2.1 attempts to categorise the refinement systems discussed in this Section according to their use of training examples, thus highlighting the following roles for training examples within the refinement context:

- driving the refinement process;
- assisting with refinement filtering; and
- enforcing a sequence on refinement implementation.

The first two roles are obvious and are commonly seen with most refinement systems. The third is less obvious, where the order in which single or subsets of examples are processed will influence refinement sequence and so the refinements that get implemented. The rows in the contingency table specify whether a single example or multiple examples can be employed with each refinement role. Clearly it does not make sense to employ a single example for the filtering and sequencing roles.

	Driving Refinement	*Refinement Filtering*	*Sequencing Refinements*
Single Example	KRUST COAST ODYSSEUS KR-FOCL CLIPS-R		
Multiple Examples	EITHER FORTE AUDREYII	KRUST CLIPS-R FORTE AUDREYII	CLIPS-R (tree structure) FORTE (refinement points) EITHER (3 stage approach) AUDREYII (2 stage approach)

Table 2.1: Contingency Table Demonstrating the Use of Training Examples.

2.2 Knowledge Refinement as Search

The generation and selection of the best refinement is a common goal for all refinement systems. With KRUST, as each example is processed in a single refinement cycle, the refinement algorithm selects the best refined KBS from a set of generated potential refinements. Essentially, a search for the best refinement through the space of possible refinements. This section introduces the Constraint Satisfaction Problem (CSP) which is a different problem to that of knowledge refinement. Approaches to solve CSPs involve search strategies that interestingly employ various heuristics that deal with search dead-ends. We are interested in investigating how knowledge refinement systems might benefit from these heuristics as a means to solve the local optima problem, provided that refinement systems are able to identify local optima when they occur.

The conventional CSP consists of a set of ordered variables $\{v_1, \ldots, v_n\}$, a finite domain D_i for each variable v_i, and a set of constraints $\{C_1, \ldots, C_m\}$, restricting the values that the variables can simultaneously take. A CSP solution is an instantiation of each variable

with a value from its respective domain, such that none of the constraints are violated (Tsang 1993). CSP search strategies fall under two broad approaches (Bartak 1999):

- the constructive approach, where solutions are sought by systematically traversing through the space of partial solutions; and
- the repair based approach of non-systematically exploring the space of complete solutions.

2.2.1 Constructive Approach

The constructive search strategy attempts to incrementally extend a partial CSP solution towards a complete solution, by instantiating one variable at a time. However, when a variable cannot be instantiated by a value from its domain without violating one or more constraints, the search strategy will fail to extend the partial solution. Essentially the search has encountered a dead-end, and terminating the search at this point is not an option, since there are variables yet to be instantiated. The backtracking paradigm is the most common algorithm for performing constructive search and dealing with dead-ends. Various backtracking algorithms have been proposed which undo the partial CSP solution, and resume the constructive process of extending the solution from a previous variable instantiation (Kondrak & van Beek 1997, Tsang, Borrett & Kwan 1994, Kumar 1992). Here, we look at three well known backtracking algorithms, that deal with the inability to instantiate variable v_j having successfully instantiated $v_1 \ldots v_{j-1}$.

Chronological Backtracking (BT) (Bitner & Reingold 1975), steps back to the most recently instantiated variable v_{j-1}, and continues the search by finding a new instantiation for v_{j-1} consistent with the current partial solution $v_1 \ldots v_{j-2}$. Upon exhausting all instantiations in v_{j-1}'s domain, BT backtracks to the next most recently instantiated variable v_{j-2}. In this manner BT recursively backtracks to previous variables, until it has identified a value for a variable, consistent with the values in the current partial solution.

BackJumping (BJ) (Gaschnig 1979), does not simply step back to the previous variable v_{j-1}, instead, it jumps back to the most recent variable v_i whose instantiation is in conflict with v_j's, and continues the search by finding a new instantiation for v_i,

consistent with the current partial solution $v_1 \ldots v_{i-1}$. If there are no new consistent instantiations available for v_i then BJ reverts to backtracking from v_i.

Conflict-directed BackJumping (CBJ) (Prosser 1993), extends the notion of backjumping by replacing the backtracking after a backjump in BJ with further backjumping if required.

```
csp-solve
    foreach v_j
        initialise(confset(v_j))
    advance(v_j)

retreat(v_j)
    if confset(v_j)= {} then
        cannot be solved
    else
        v_i is variable in confset(v_j)
            with highest subscript
        confset(v_i) := confset(v_i) ∪ confset(v_j) \ {v_i}
        for N = i + 1 to j
            initialise(confset(v_N))
            initialise D_N to original domain values
        if D_i = {} then retreat(v_i)
        else advance(v_i)

advance(v_j)
    foreach d_jk in D_j
        remove d_jk from D_j
        foreach v_i in v_{j-1}, ..., v_1
            consistent := true
            foreach C in C_1, ..., C_m
                if v_j=d_jk and v_i violates C then
                    consistent := false
                    exit foreach C
            if ¬consistent then
                update(confset(v_j),v_i)
                exit foreach v_i
        if consistent then
            exit foreach d_jk
    if consistent then
        if j < n then advance(v_{j+1})
        else solved
    else retreat(v_j)
```

Figure 2.3: A General Backtracking CSP Algorithm.

An algorithm that finds the first possible solution to a CSP is shown in Figure 2.3. This algorithm is applicable to binary CSPs where a constraint involves at most 2 variables. Here, each backtracking algorithm applies the generic **csp-solve** function. Associated with each variable is a *conflict set* (confset) of potential backtracking points. The initialisation and update of the confset is defined as follows:

$$\textbf{initialise}(confset(v_j)) = \left\{ \{\} \quad \text{for BT, BJ, CBJ} \right.$$

$$\textbf{update}(confset(v_j), v_i) = \begin{cases} \{v_{j-1}\} & \text{for BT} \\ confset(v_j) \cup \{v_1, \ldots, v_i\} & \text{for BJ} \\ confset(v_j) \cup \{v_i\} & \text{for CBJ} \end{cases}$$

When advancing search, a value d_{jk}, from v_j's domain D_j, is selected such that instantiating v_j with value d_{jk} will not violate any of the constraints. The process of identifying a consistent instantiation value for v_j, involves trying each value d_{jk} and examining whether a previously instantiated variable, v_i, violates any of the constraints. If a violation occurs then the value is discarded and the next value in D_j is tried. The v_i that was involved in the violation gets added as a potential backtrack point when v_j's confset is updated. The update of BT's confset is trivial, as it always contains the previous variable as the potential backtracking point. BJ updates v_j's confset with the conflicting v_i, together with all variables preceding v_i, as potential backtracking points. CBJ is similar to BJ, but only v_i is maintained as a backtracking point. This enables CBJ to recursively back-jump to previous conflict points, because unlike with BJ any intermediate variables are not maintained. It is in the **retreat** function that these subtle differences of updating the confset come into play, enabling the implementation of three distinct backtracking policies within the generic csp-solve function. Eventually, if v_j cannot be instantiated because all values from D_j are exhausted, then backtracking is necessary and this responsibility is passed on to the retreat function.

When search has to retreat, the variable with the highest subscript (or most recently instantiated) is always selected as the backtrack point. Any inconsistencies experienced when advancing the search are recorded with the update of the confset. The union of the confset for v_j (the variable at the dead-end point) with the confset for v_i, ensures that these experiences are not forgotten even when search has to resume from a previous stage. However, it is only with CBJ that such explicit knowledge about past experiences is exploited, as here the updating of confset maintains v_i as the backtracking point. Accordingly, if D_j is $\{\}$, CBJ will be able to reuse any past knowledge about inconsistencies, hence the potential to backjump further.

The confset variables identified during a search can also be used as an opportunity for learning, in addition to focusing backjumping. The technique of learning in this manner is called constraint recording and can also be viewed as explanation based learning (Dechter & Frost 1999). The idea is to add the contents of the confset (no goods) in the form of new constraints. This increases search space pruning, whereby the same inconsistencies will not be rediscovered. Kambhampati's (1998) approach to planning, exploits the ideas of backjumping and explanation based learning. Although the planning activities and their

sequences are not fixed, the descriptions of each activity contain static information specifying preconditions and effects of the activity. When inconsistent activities are identified, their preconditions and effects are noted and formulated into new constraints.

2.2.2 Repair-Based Approach

The repair-based search for a CSP solution guesses an initial solution to the CSP that is likely to be inconsistent. This solution is then incrementally altered by changing values of strategically selected variable instantiations (Bart Selman & Cohen 1994). Typically, the selection strategy adopts a hill-climbing approach that advances to the next best state. Such a strategy is computationally expensive, as all variables and possible value instantiations must be explored. The min-conflicts heuristic aims to reduce this exploration task by selecting a random inconsistent variable, and instantiating it with the value that conflicts least with the rest of the variables (Minton, Johnston, Philips & Laird 1992). However, the hill-climbing repair based approach suffers from the common local optimum problem, and must be complemented with randomised techniques, that can get out of and beyond, the local optimum. The mixed random-walk strategy (Selman & Kautz 1993), introduces controlled randomness to the search by interspersing a random step with probability p, and the hill-climbing step with probability $1-p$. During the random step, an inconsistent variable is randomly picked and its instantiated value is changed. Another approach maintains a short-term memory of past actions (a tabu list), and ensures that the same past actions are not repeated within a specified tenure (Glover & Laguna 2000), thus avoiding repetitive actions.

The constructive CSP approaches we have considered, are well equipped to handle dead-ends and can efficiently get beyond local optima. Although repair-based search must be equipped with randomised techniques to handle local optima, it has been shown that these search strategies operate more efficiently in large search spaces where solutions are not evenly distributed (Minton et al. 1992). One contributory factor towards this improved efficiency, is due to the use of information about the current solution by repair-based search, not available to constructive search. This is because, repair-based search deals with a complete yet inconsistent solution, while constructive search deals with an incomplete yet consistent partial solution.

2.2.3 Improving Search Efficiency

CSPs employ various heuristics that reduce search effort (Frost & Dechter 1994, Sadeh & Fox 1990).

- Value ordering heuristics select those values that conflict least with variables that are yet to be instantiated;
- Variable ordering heuristics deal with most constrained variables first.

Value ordering aims to select a variable instantiation that is most likely to lead to a solution without the need for backtracking. The repair-based approach to solving CSPs and its greedy min-conflict heuristic for repair selection (Minton et al. 1992) is an effective heuristic of this kind. An interesting value ordering heuristic that can be employed to improve constructive search efficiency with close parallels to Minton et al.'s min-conflict heuristic is the look-ahead min-conflict value ordering heuristic for constructive search (Frost & Dechter 1995). Here, values of a variable are ranked in increasing order based on the number of incompatibilities with potential values of future variables to be instantiated. For the generic **csp-solve** function in Figure 2.3, this would mean a simple modification to **advance**: sort the possible instantiation values in D_j in ascending order, according to the number of incompatibilities with all potential values of variables yet to be solved, $\{v_{j+1}, \ldots, v_n\}$.

The intuition behind variable ordering heuristics is to deal with the most constrained variable first, thereby enabling early discovery of dead ends, hence efficient pruning of the search space. A CSP variable is generally constrained in two ways:

- by the constraints it is involved in; and
- by its domain size.

Most common variable ordering heuristics exploit these two properties separately or in combination (Dechter & Meiri 1994, Gent, MacIntyre, Prosser, Smith & Walsh 1996, Gent, MacIntyre & Prosser 1996, Brelaz 1979). Essentially, these heuristics aim to deal with variables involved in the most number of constraints and/or, with smaller domain sizes first. Heuristics for static variable ordering exploit relationships among variables identified from the topology of the constraint graph (Tsang 1993). Here, the aim is to

deal with tightly constrained variables early, consequently reducing the number of revisits to previously instantiated variables. Dynamic variable ordering addresses the fact that invariably the best variable order is different in different branches of the search tree, by taking advantage of the information available after each variable instantiation to move the search to branches that are more likely to contain a solution (Haralick & Elliott 1980). Various look-ahead strategies select the variable that most constrains the remainder of the search (Smith & Grant 1998). The motivation behind all such heuristics is to deal with variables that are most constrained first since leaving them until later can only lead to increased demands on consistency and late discovery of dead-ends.

2.2.4 Dynamic CSPs

Recent developments in constraint programming have increased pressure to adopt CSP techniques for scheduling and planning tasks (Bartak 2000). The main difficulty with these tasks involve changing variables, domains and constraints during search. These difficulties have warranted the extension of the conventional CSP framework to a Dynamic CSP (DCSP) framework, in which variables, domains and constraints are not required to be known fully in advance (Mittal & Falkenhainer 1990, Lamma, Mello, Milano, Cucchiara, Gavanelli & Piccardi 1999, Kambhampati 1998).

Mittal & Falkenhainer (1990) introduce the idea of active variables and non-active variables. The solution to a CSP involves instantiating all active variables. However, as a result of instantiating an active variable, non-active variables can become active depending on the value being instantiated. This necessitates a new type of constraint called *activity constraint*, which constrains a variable to be active or not active, based on other variables that are active and on their value instantiations. For instance, given a car configuration task, if the frame of the car is to be a convertible, we need to gather information about the types of sun roofs, if however the frame is simply a hatch-back, then the variable associated with sunroof need not be active. Additionally, standard constraints are distinguished from activity constraints as *compatibility constraints*, because of the active/non active variable distinction. However unlike standard constraints, compatibility constraints are applicable only if all variables that are constrained by it are active. Essentially, the idea of activity constraints extends standard CSPs to handle a changing set of variables and reasons about the activity of variables. The idea of activity constraints is adopted

for solving planning tasks in (Kambhampati 1998, Kambhampati 2000, Kambhampati & Nigenda 2000). Here, as the initial state of a plan is incrementally transformed into the goal state via intermediate states triggered by the application of plan actions, the effects of these actions can necessitate the application of other actions. For instance, given an object, with the goal of shaping and polishing it into a cylinder, then rolling the object to make it cylindrical, will necessitate the application of the new action of cooling the object, as polishing cannot take place when the object is heated. Therefore the effects of plan actions are analogous to activity constraints, but there is no direct analogue to compatibility constraints.

For constraint problems where the acquisition of all domain values is not convenient, the idea of interactive constraints is introduced in (Lamma et al. 1999). Interactive constraints in addition to standard constraints, can also constrain variables for which all domain values are not known. A variable gets instantiated with values from its known domain, if however the domain is completely unknown, then value acquisition is triggered and guided by any unary constraints. Forward checking involves constraint propagation to ensure that future variables are consistent with the newly instantiated variable. During the process of forward checking, with the interactive CSP approach, if domains of any future variables involved in a constraint are unknown then value acquisition is initiated. Therefore, interactive constraints extend the role of standard constraints, to cover variables with domains partially known or even completely unknown, and triggers and guides value acquisition during forward checking.

2.3 Example Selection for Learning Algorithms

Fundamental to the operation of all refinement systems in Section 2.1 is the availability of labelled examples. Surprisingly, all refinement systems assume that a representative set of labelled examples will be available, and do not address the potential problem of refinement constrained by the availability of labelled examples. Although a batch version of EITHER has been implemented to deal with the availability constraint it assumes that examples contained within batches that become available are all labelled (Mooney 1992). In a real environment unavailability of labelled examples is a relatively common problem, where labelling many problem-solving tasks with the expert's solution may require significant

interaction with a busy expert.

The problem of unavailability of labelled training examples and sample selection of relevant examples from a set of unlabeled examples, has been addressed by machine learning algorithms and falls under the paradigm of active learning. There are two main approaches to active learning:

- membership queries, in which the learner constructs examples and asks an expert to label them (Angluin 1988, Angluin, Frazier & Pitt 1992); and
- *selective sampling*, where the learner examines many unlabelled examples, selecting only the most informative ones to be labelled by an expert.

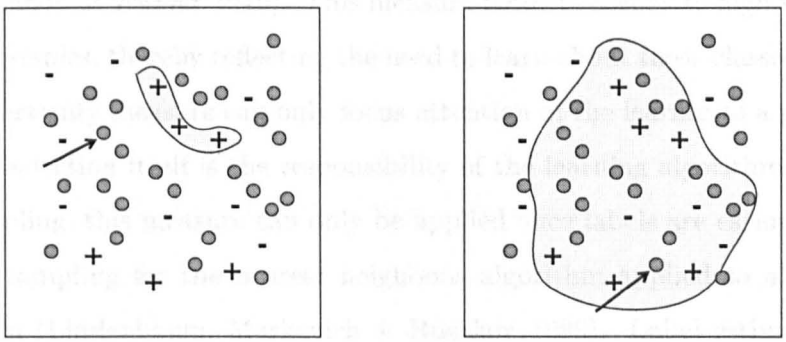
Here, the focus of interest is in selective sampling, where the underlying assumption is that a large set of unlabelled training examples are available. Such an assumption is not unreasonable, as unlabelled examples can be generated based on meta knowledge (Zlatareva & Preece 1994, Ayel & Vignollet 1993). Moreover, in some problem domains the observations are readily available but the labelling is costly, e.g. document classification on the web.

Selective sampling strategies utilise information from labelled examples to perform selection on the unlabelled examples. Typically, the labelled examples are used to train a classifier which then predicts the labels of those unlabelled, and selection exploits classification uncertainty of the classifier, i.e. selects those examples for which the current best model is most uncertain. More recently, this approach has been extended to include several classifiers that operate as a committee, and selection exploits classification uncertainty reflected by the committee as a whole, thereby evaluating classification uncertainty with respect to a subset of models from the entire space of possible models.

2.3.1 Uncertainty-Based Classifiers

Cohn, Atlas & Ladner (1994) apply selective sampling to the task of learning a binary concept, by using a specific-general (SG) neural network configuration. The specific configuration (S-net), is achieved by preferring a network configuration with highest accuracy on a data set, formed by arbitrarily adding unlabelled examples and treating them as negative examples. Essentially, this is training with a negative bias, where the S-net covers

a few positive examples clustered in a small area of the version space. The most general configuration (G-net), is formed by introducing a positive bias resulting in the G-net covering all positive examples, but in doing so will also include negatives. Figure 2.4 illus-



(a) (b)

Figure 2.4: Learning with (a) a Negative Bias and (b) a Positive Bias.

trates for a binary classification problem, the different generalised and specialised concept boundaries learned by the G-net and S-net. The dots in Figure 2.4(a), represent negatively labelled examples arbitrarily added to create a negative bias, and in Figure 2.4(b), positively labelled examples to create a positive bias. The concept is indicated by the shaded area. The objective is to exploit the region of uncertainty, by selecting unlabelled examples for which both network configurations fail to agree upon their labels. The arrows in Figure 2.4, highlight two examples for which the S-net and G-net are bound to disagree. An example in the uncertain area once selected, is labelled by the expert, consequently the network configuration is updated as follows:

- If the S-net had incorrectly estimated the example as negative, then the S-net concept boundary is generalised to cover these examples; otherwise
- if the G-net had incorrectly estimated an example as positive, then the G-net concept boundary is specialised, such that it does not cover this example.

The uncertainty-based example selection and labelling process iterates until both network configurations converge. An obvious danger in this approach is that with complex concepts the G-net may cover the entire version space, reducing the efficiency of selective sampling to that of random sampling. Therefore, an obvious difficulty with this approach is in identifying the initial concept boundaries specially for domains with multiple

classes and complex concept boundaries. The inductive learning algorithm DIDO (Scott & Markovitch 1989) also employs informative example selection at concept boundaries, and avoids the boundary identification problem by employing the Shannon uncertainty measure (Shannon & Weaver 1949). This measure tends to associate high scores to classes with fewer examples, thereby reflecting the need to learn about these classes. However, the Shannon uncertainty measure can only focus attention of the learner to a particular class, and example selection itself is the responsibility of the learning algorithm. Therefore, for selective sampling, this measure can only be applied once labels are estimated.

Selective sampling for the nearest neighbour algorithm applied to a binary domain is discussed in (Lindenbaum, Markovich & Rusakov 1999). Label estimation of an unlabelled example is done by means of a linear combination function, that combines the classes of labelled examples according to correlation between feature vectors. The result of this correlation is that labelled examples that are also the nearest neighbours, tend to influence the label estimation of an unlabelled example. The greater influence is reflected by higher weights assigned to nearest neighbours. The weights are obtained by matrix transformation employed when fitting a multiple regression model (Mendenhall & Sincich 1988). However, one problem with using a linear combination function as a label estimator is that class estimates can result in values outside the binary class range $[0, 1]$, requiring some form of rounding.

Once class labels have been estimated for all unlabelled examples, the next task is to select D' examples that would increase the accuracy of the nearest neighbour classifier. A utility function is employed for this purpose comparing the gain in classification accuracy over the unlabelled examples U, between the classifier h formed using the labelled examples D, and the classifier h' formed using $D \cup D'$. The D' that increases accuracy is selected for labelling. The gain in accuracy can be a simple comparison of how many examples in U were correctly classified by h and h' according to the nearest neighbour principle. However, Lindenbaum et al. employs h and h' as probabilistic classifiers. The experimental results are encouraging, although the test domains have been restricted to binary classification tasks.

Lewis & Catlett (1994) employ a probabilistic classifier to select examples for the C4.5 learning algorithm (Quinlan 1993). Once trained on labelled data, the probabilistic classifier is applied to the unlabelled examples, selecting those about which it is most

uncertain. The motivation here is to use the cheaper probabilistic classifier to select examples for the relatively expensive C4.5 classifier. Experimental results suggest that labelling cost was reduced when example selection was carried out by the probabilistic classifier and by the more expensive C4.5 algorithm. However, there is some penalty on classifier accuracy when uncertainty sampling is carried out by the probabilistic classifier for C4.5, instead of C4.5 itself.

2.3.2 Committee of Uncertainty-Based Classifiers

Freund, Seung, Shamir & Tishby (1997), employ a two-member committee drawn from a sample of labelled examples to select informative examples that can then be labelled by an expert. An informative example is one for which the committee fails to agree about its label, because the example has changed the representation of the hypothesis. Generating a committee member involves selection of parameter values that are required for class probability estimation. These parameter values are selected according to the underlying statistics of the labelled examples. Essentially, a member can be viewed as a set of parameters that are needed to estimate class labels of unlabelled examples given their features. For each unlabelled example, each member in turn estimates its label. Informative examples are selected according to committee disagreement and labelled by the expert. The newly labelled examples will now influence the parameter values, therefore, a new committee is generated according to the underlying statistics. This process can continue until consecutive agreement between the committee is above a predetermined threshold. Notice that with an increasing number of labelled examples the variance between parameter values picked for committee members decreases, thus leading to committee members with fewer disagreement.

The basic two member committee approach is extended to k members in (Argamon-Engelson & Dagan 1999). Here, the committee-based approach is applied to learning Hidden Markov Models (HMMs) (Merialdo 1991) for part-of-speech tagging of English sentences. Part-of-speech tagging involves labelling each word in a sentence according to its role in the sentence, e.g. verb, noun etc. HMMs are trained so that given a sentence it is able to classify the words into their respective sentence roles. The committee-based approach is employed to aid the HMMs to learn efficiently achieving improved accuracy using fewer training examples. Unlike with the two member committee, with k members

a more sophisticated measure of disagreement is necessary. The *vote entropy measure*, captures the uniformity in class estimation for an example by the different committee members. Given the set of classes C, and the number of committee members classifying e in class c, where $c \in C$, denoted by $votes(c, e)$, the vote entropy is:

$$vote\ entropy(e) = -\frac{1}{log\ min(k, |C|)} \sum_{c \in C} \frac{votes(c, e)}{k} log \frac{votes(c, e)}{k}$$

The entropy measure is normalised by a bound on its maximum possible value, log $\min(k,|C|)$, such that the value is between 0 and 1. The higher the value the greater the disagreement.

An alternative measure of disagreement is suggested by McCallum & Nigam (1998), using the *Kullback-Leibler (KL) divergence to the mean.* Here, the measure relies on the probability attached by each committee member, that an example is in c. Therefore, unlike the vote entropy, which compares only the committee members top ranked class estimates, KL divergence can measure the strength of the disagreement based on all class distributions. Although the vote entropy measure is appealing due to simplicity, experiments indicate that KL divergence results in improved accuracy when sampling on domains with sparse examples. The improved performance of KL divergence is explained by its ability to select examples that were sufficient to learn generalised concepts. Interestingly, this means that the vote entropy measure tends to select atypical examples, while the KL divergence avoids atypical examples. Therefore, for applications where the aim is to learn new concepts it makes sense to avoid atypical examples, as they would lead to skewing of statistics; whereas, for the task of identifying difficult or noisy examples, it makes sense to concentrate on atypical examples.

An interesting approach combining a committee with Expectation Maximization (EM), is employed for the task of text classification in (McCallum & Nigam 1998). A common difficulty for text classifiers is the need for a large, often prohibitive, number of labelled training documents to learn accurately. The task of training a text classifier using a limited number of labelled documents is tackled in (Nigam, McCallum, Thrun & Mitchell 1998, Nigam, McCallum, Thrun & Mitchell 2000), by incorporating information from unlabelled documents. The labelled documents are used to calculate the initial parameter

values, which consist of:

- the class probability distributions for each word, i.e. the number of word occurrences for a given class normalised by all word occurrences in that class; and
- the prior probability distributions for classes, which is the ratio of documents in a given class.

These parameters are then used by a naive Bayes classifier (Domingos & Pazzani 1997) to estimate the most likely label for the set of unlabelled documents. The labelled documents and the newly labelled (estimated) documents are then used to form new parameters values. This process iterates until there are no changes (or minimal changes) to the parameter values between consecutive iterations. Unlike selective sampling where labels can be sought from an expert, here the learning relies on the Bayes estimates and on the convergence of parameter values. Essentially, this process of estimating labels and incorporating the newly labelled examples in the re-calculation of parameters for the next iterations, falls under the Expectation Maximization class of algorithms (Dempster, Laird & Rubin 1977).

Although text classification with EM is able to improve classification accuracy by supplementing scarce labelled documents with unlabelled documents, selective sampling has the potential to improve accuracy further, as labels for selected examples can be sought from the expert (McCallum & Nigam 1998). The committee based approach is extended to include EM, where each committee member applies EM before estimating the final set of labels for the unlabelled examples (see Figure 2.5). A member is generated by randomly selecting parameter values constrained by the variance. For instance, the variance for the parameter $p(t|c)$ that estimates the probability of word t given class c, where the total number of word occurrences for c is n, is specified as:

$$\frac{p(t|c)(1-p(t|c))}{n}$$

Here, the variance decreases as more labelled examples are available for the parameter estimation. The result is that labelling of selected examples improves the estimation of parameters for EM, while incorporating EM with each committee member, avoids selecting examples whose labels can be reliably predicted by EM.

1. Calculate initial parameter values θ, and variance σ, for these values based on labelled documents.

2. For i = 1 to k
   (a) Randomly select θ_i from the range constrained by θ and σ.
   (b) Initialise new θ_i as θ_i
   (c) Apply EM with the unlabelled documents and Repeat:
       * Let θ_i = new θ_i
       * Estimate labels for unlabelled documents using θ_i.
       * Calculate new θ_i based on the labelled and newly labelled documents.

       Until θ_i = new θ_i
   (d) Use new θ_i to estimate labels for all unlabelled documents.

3. Calculate the disagreement between the k members based on the label estimates in step 2 (d), and request labels where disagreement is high.

Figure 2.5: Complementing committee based approach with EM.

The use of a committee to identify and eliminate noisy examples, instead of selecting examples, is discussed in (Brodley & Friedl 1996). Here, the goal is to improve the quality of the training set consisting of all labelled examples but possibly contaminated with noise. Essentially, the committee acts as a filter that identifies and eliminates noisy examples from the training set. A committee with k members is generated by performing a n-fold cross-validation over the training examples (see Figure 2.6). Each member is a learning algorithm that is trained on the $1, \ldots, n-1$ parts of the training set. The resulting classifier estimates the class of each example in the nth part; if the estimate is correct, the example is tagged as correct otherwise tagged as mislabelled. The tagging by the committee is analysed to establish whether the example should be eliminated or not. The consensus heuristic requires complete consensus between members before an example is retained. The less conservative majority heuristic requires that a majority of members are able to classify the example correctly before it can be retained. With both heuristics there is the danger of either being too conservative and retaining noisy examples or of detecting noisy data at the expense of throwing away useful examples. Therefore, when training data is scarce it makes sense to employ the majority heuristic, and for situations where examples are in abundance the more conservative approach is suited.

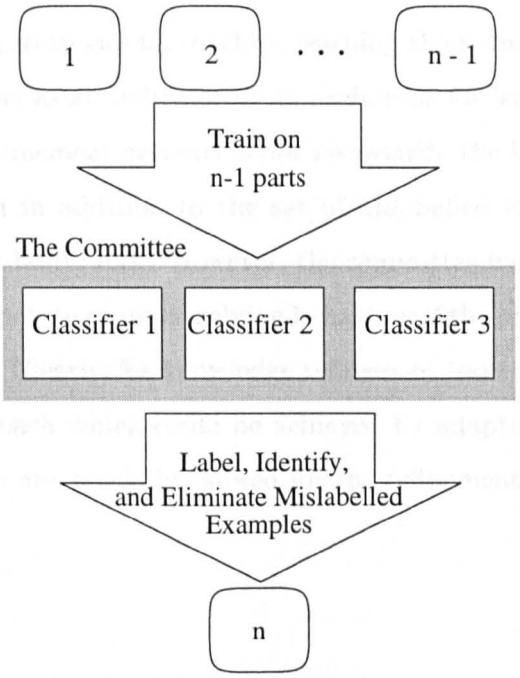


Figure 2.6: Filtering Noisy Examples Using a Committee of Classifiers.

2.4 Conclusion

Refinement systems employ training examples to drive the refinement process. Additionally, training examples are useful for refinement filtering and refinement sequencing. Ideally, a refinement system should employ training examples for all three purposes: driving, filtering and sequencing. The main weakness with most refinement systems is the hill-climbing based refinement search approach which invariably results in the local optima problem. Therefore, we need search mechanisms that can help guide refinement by avoiding local optima, and recovery strategies that help deal with local optima when they occur.

Search strategies for CSP show that backtracking strategies are commonly employed to deal with search dead-ends. A CSP search dead-end can be viewed as a local optimum, because the search for a consistent CSP solution requires getting out of the dead-end, and moving to a different part of the search space. However, an analogy between CSP search and refinement search is required before CSP search and ordering strategies can be incorporated by a refinement system.

The choice of training examples for refinement becomes important when one of the

constraints on the refinement process is a limited number of labelled training examples. Most selective sampling strategies for machine learning algorithms, exploit the uncertainty about an example's label as an indicator of its usefulness for learning. Direct application of these strategies to refinement systems is not necessarily the best way forward, because refinement systems can in addition to the set of unlabelled examples, exploit problem solving behaviour of the faulty KBS. However, the committee based approach may provide an opportunity to incorporate problem solving behaviour if the proposed refined KBSs were to form the committee. Clearly, for knowledge refinement tools we need to incorporate an example selection approach which could be achieved by adapting existing approaches or by new approaches that are specially catered for the refinement task.

Chapter 3

Iterative Refinement

An iterative approach to refinement aims to incrementally improve the accuracy of the refined KBS with each refinement cycle. Occasionally, contrary to such expectations, a greater gain in accuracy can be achieved in subsequent iterations, by deliberately undoing incremental effects in preceding iterations. The emphasis of this chapter is on the use of training examples to guide and direct the KRUSTtool's refinement process through the space of possible refinements. Essentially, we are interested in improving the efficiency when searching for refinements, and for this purpose, various search techniques for solving constraint satisfaction problems are adapted for iterative refinement, with a view to improving accuracy in the final refined KBS. The results reported in this chapter have also been published in (Wiratunga & Craw 1999a).

A general iterative refinement framework is presented in Section 3.1. A solution to the obvious drawbacks in this framework leads to the discussion in Section 3.2 which presents an analogy between CSPs and iterative refinement as a way to incorporate CSP search strategies for iterative refinement. Section 3.3 analyses differences between CSPs and iterative refinement and discusses how CSP search strategies might be adapted for the KRUSTtool and the implications of these adapted search strategies on iterative refinement is presented in Section 3.4. An initial study of re-ordering techniques which address refinement search efficiency is presented in Section 3.5, before concluding with Section 3.6.

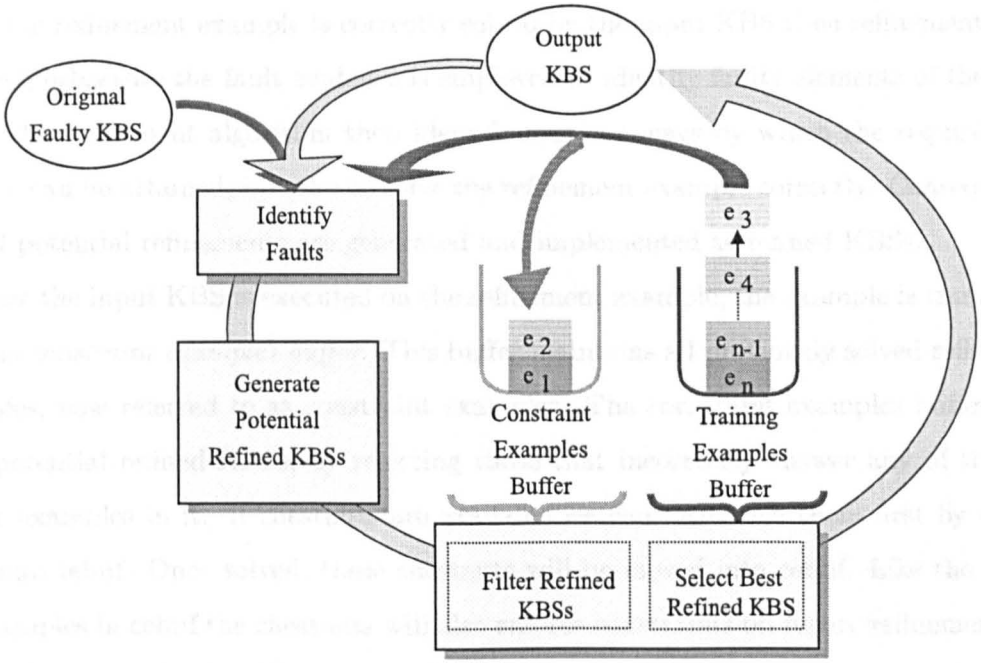


Figure 3.1: The iterative refinement process.

3.1 The Iterative KRUSTtool Process

The KRUSTtool can be employed to carry out iterative refinement as illustrated in Figure 3.1. The *input KBS* is the best refined output KBS from the previous iteration, or the original faulty KBS for the first iteration. The *training examples buffer* contains the training examples $\{e_1, \ldots, e_n\}$, which are utilised one at a time. Each training example e, is a task-solution pair $\langle [f_1, \ldots, f_m], goal \rangle$; where the observables $f_1, \ldots, f_m$ are the facts that initialise the problem-solving task, and its solution *goal* is the example's label acquired from the expert. For each iteration, the top most example in the training examples buffer is chosen as the refinement example and drives that refinement cycle.

The refinement example's observables initialise the problem-solving task for the input KBS, which triggers the KBS's reasoning process, resulting in the system solution. The input KBS:

- solves the refinement example correctly, if the system solution matches the example's solution; or
- solves the refinement example incorrectly, if there exists a mismatch between the system's solution and the example's solution.

When the refinement example is correctly solved by the input KBS then refinement is not required, otherwise the fault evidence is employed to identify faulty elements of the input KBS. The refinement algorithm then identifies various ways by which the required goal solution can be attained, in order to solve the refinement example correctly. Consequently, several potential refinements are generated and implemented as refined KBSs.

Once the input KBS is executed on the refinement example, the example is transferred into the *constraint examples buffer*. This buffer maintains all previously solved refinement examples, now referred to as constraint examples. The constraint examples buffer helps filter potential refined KBSs, by rejecting those that incorrectly answer any of the constraint examples in it. If chestnuts are available we can deal with them first by adding them into tebuf. Once solved, these chestnuts will be moved into cebuf. Like the rest of the examples in cebuf the chestnuts will also enforce constraints on future refinement iterations. Filtered refined KBSs that are consistent with cebuf are ranked by their accuracy on the remaining examples in the training examples buffer, and the refined KBS with the highest accuracy is the output KBS for this iteration. The iterative refinement process continues until an output KBS that correctly solves all examples in the training examples buffer is produced, this KBS is the final output KBS.

The hill-climbing selection of the *one* best refined KBS for the next iteration occurs at the *end* of each cycle. This selection works well provided it is possible to select, from the set of generated refined KBSs an output KBS with accuracy always greater than that of the input KBS. However, it is not unusual to have a refinement cycle where the KRUSTtool fails to generate any refined KBSs, or where the input KBS has greater accuracy than any of the generated refined KBSs. Such complications are symptoms of the local optimum problem, common to hill-climbing search algorithms. Once a local optimum is reached, further advancement does not lead to any improvement in accuracy, and here we refer to this situation as a *refinement dead-end*. Terminating search when a dead-end is encountered is not an option, particularly when there are examples yet to be processed in the training examples buffer. Fortunately, the KRUSTtool's iterative refinement algorithm, generates and implements several refined KBSs in each refinement cycle, hence, the opportunity to explore previously abandoned refinement alternatives.

Figure 3.2 illustrates the start of a potential backtracking scenario; the updates to the constraint examples buffer (cebuf) and the training examples buffer (tebuf) are shown

on the right. Each node denotes the refinement example in a given iteration. An arc connecting consecutive refinement examples, denotes the best refined output KBS from $iteration_i$ being used as the input KBS in the subsequent $iteration_{i+1}$. All abandoned refined KBSs in an iteration are indicated by open ended arcs. For instance, in $iteration_2$, three refined KBSs are generated with e_2 as the refinement example. All three refined KBSs are consistent with examples in cebuf, but K_{21} with highest accuracy on tebuf, is selected as best and is the input KBS in $iteration_3$, while K_{22} and K_{23} are abandoned. Refinement examples e_3 and e_4 have triggered the generation of several refined KBSs and again the best is selected. In $iteration_5$, we have a situation where, K_{41} cannot be refined by e_5, because, although four refinements are generated, all are rejected by the constraint examples in cebuf. Here, the KRUSTtool has reached a dead-end, indicated by a darkly shaded node for e_5, with four dashed lines corresponding to the four generated refinements. The *refinement path* consists of a series of incremental refinements made to the original faulty KBS; in the diagram this path is $\ldots\ e_2 \xrightarrow{K_{21}} e_3 \xrightarrow{K_{31}} e_4 \xrightarrow{K_{41}} e_5.\emptyset$ where $\emptyset$ indicates the absence of a selected refined KBS. Strictly, it is this path that labels the refined KBSs in the diagram and so the output KBS labelled K_{51} is really named $K_{\ldots 21314151}$, which is the outcome of consecutive refinements K_{21}, K_{31}, K_{41} and K_{51}.

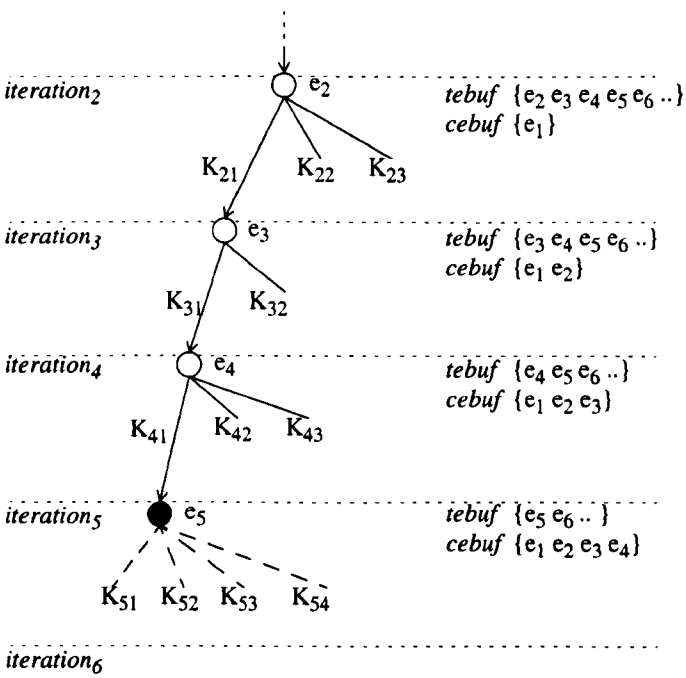


Figure 3.2: Knowledge refinement as search.

When the iterative refinement process encounters a dead-end, as in $iteration_5$, the following alternative strategies might be employed to get out of, and beyond the dead-end:

- continue with K_{41} to $iteration_6$ (the next iteration) with e_6 as the refinement example, ignoring the fact that e_5 is not corrected, and is unlikely to be by future refinements; or
- continue with the refined KBS selected from $K_{51}, \ldots, K_{54}$, with the highest accuracy on tebuf, ignoring the fact that the selected refined KBS is inconsistent with one or more previous examples in cebuf; or
- generate a fix that explicitly solves e_5 only, even though in practice KBSs with such refinements may be too specialised.

A much more desirable alternative is to backtrack through the solution space of refined KBSs, and restart the refinement process with a rejected refined KBS from an earlier node. For the scenario in Figure 3.2, this would entail undoing the most recent successful refinement (which was in $iteration_4$), and restarting refinement with K_{42} and e_5. Backtracking in this manner although simple might not necessarily be efficient, as the actual cause of the dead-end could be further up the refinement path. Therefore, we investigate various search strategies that enable more guided backtracking. For iterative refinement, this would mean the ability to restart the refinement process from an earlier point responsible for the current dead-end.

3.2 CSP Search Strategies for Knowledge Refinement

Incorporating CSP search strategies with the iterative KRUSTtool, will enable efficient search through the space of incrementally refined KBSs generated by the KRUSTtool. Consequently, the KRUSTtool, when necessary, will be able to revisit refined KBSs that have previously been abandoned by the refinement algorithm. For this purpose we need to propose an analogy between CSPs and knowledge refinement problems, so that the concepts applied in the CSP paradigm can be imitated in the refinement domain.

CSP	Iterative KRUSTtool
Variables	Input/Faulty KBSs
Current Variable (v_i)	Current Input KBS (K_i)
Instantiated Variables	Output/Refined KBSs
Uninstantiated Variables	Unrepaired Faults
Variable Domain for v_i	Proposed Refinements for K_i
Constraints	Consistency with cebuf

Table 3.1: Analogy 1: CSPs and Iterative Refinement.

3.2.1 Knowledge Refinement as Constraint Satisfaction

Traversing the space of possible solutions efficiently and discovering a suitable solution, are common goals of both CSP search algorithms and refinement algorithms. One possible analogy relates CSP variables to input KBSs, instantiated variables to output KBSs and constraints to maintaining consistency with cebuf (see table 3.1). However, the absence of relating training examples in some manner to refined KBSs, will cause problems when backtracking is triggered. For instance, how does one identify which example to execute the input KBS soon after a back track or a back jump. Furthermore, as there is no prior knowledge about the number or type of faults in an input KBS, there is no obvious analogy for uninstantiated variables.

The dynamic nature of the refinement task makes it more complex than the well defined CSP task, however, a second analogy in Table 3.2 gives prominence to the training examples, hence, a more static view of refinement search. The training examples, like the variables, are fixed from the onset of the refinement task. Constraint examples in cebuf, are solved correctly or have been corrected in the past and so are analogous to instantiated variables. The best refined KBS is selected from the set of potential refined KBSs generated by the refinement algorithm in a refinement cycle. Therefore, proposed refined KBSs are comparable to the variable domain, although refined KBSs become known only with each refinement cycle, unlike the variable domain which is fixed from the start. However, the problem of not knowing the domain in advance can be handled by associating generated refined KBSs with the refinement examples that triggered the refinement cycle, and reasoning about backtracking using constraint examples rather than KBSs. Finally, CSP constraints are analogous to maintaining consistency with cebuf. However, unlike well defined CSP constraints that are known in advance, for knowledge refinement there are no obvious initial constraints that specify mutually incompatible refinements.

CSP	Iterative KRUSTtool
Variables	Training Examples
Current Variable (v_i)	Refinement Example (e_i)
Instantiated Variables	Constraint Examples in cebuf
Uninstantiated Variables	Remaining Examples in tebuf
Variable Domain for v_i	Generated Refined KBSs for e_i
Constraints	Consistency with cebuf

Table 3.2: Analogy 2: CSPs and Iterative Refinement.

With many applications it is not possible to acquire all domain values at the beginning, as the acquisition process can be computationally expensive (e.g. 3D object recognition), or the domain values can be unavailable at the beginning (e.g. knowledge refinement). The Interactive CSP framework proposed by Lamma et al. (1999) attempts to deal with this problem by introducing the idea of interactive acquisition of domain values on demand or when made available. However, with this approach it is assumed that all constraints are specified in advance enabling value acquisition on-demand, guided by constraints. With knowledge refinement we are unable to adopt this interactive approach, because in addition to unknown domains, there is also no prior knowledge about mutually incompatible refinements.

3.2.2 Informed Backtracking with the KRUSTtool

Using the second analogy from the previous Section we investigate how backtracking might be applied to iterative knowledge refinement. An **advance** (see Figure 2.3) with refinement search is triggered by executing the input KBS on the next refinement example e_i. When e_i is incorrectly solved by input kBS, the KRUSTtool generates several potential refined KBSs, $Generated_i$. The generated KBSs can be viewed as e_i's domain for this iteration. Of these generated refined KBSs, those that are inconsistent with constraint examples in cebuf are rejected. The remaining subset of refined KBSs, $Filtered_i$, are sorted in descending order of accuracy on training examples yet to be processed in tebuf, and the refined KBS with the highest accuracy is selected.

Notice that a generated refined KBS from a refinement example's domain is inconsistent when it fails to solve a constraint example correctly, interestingly, this is analogous to a variable instantiation that violates a binary constraint with CSPs. The sorting and selecting step, corresponds to that part of the informed backtracking algorithm which aims

to improve efficiency by sorting of values according to conflicts with available values of future variables.

The conflict set for e_i, $confset(e_i)$, must contain the potential backtracking points from e_i, which is exploited when ever a dead-end is encountered. For this purpose the confset of every refinement example needs to be updated, and takes place immediately after $Filtered_i$ is identified. Once $Filtered_i$ is known, we will also know which constraint examples caused the removal of each generated KBS. These inconsistent constraint examples are noted by adding them into $confset(e_i)$ according to the **update** policy in Section 2.2.1. Obviously, when $Generated_i$ is identical to $Filtered_i$, $confset(e_i)$ will be empty.

The search algorithm encounters a dead-end when a refinement example e_i and the input KBS fail to create any refined KBSs (i.e. the generated KBSs $Generated_i$ is empty), or those generated are rejected by the constraint examples (i.e. $Filtered_i$ is empty). The updated $confset(e_i)$, will contain the potential backtracking points from e_i. We now consider two different scenarios and how dead-ends are dealt with in each.

- If $Filtered_i = \{\}$ then we know which constraint examples caused the removal of each generated KBS, and the **retreat** function in Section 2.2.1 will be called with the relevant backtracking point according to the various informed backtracking algorithms (i.e. BT, BJ and CBJ).

- If however, $Generated_i = \{\}$ then conflicting constraint examples cannot be identified since there are no KBSs to test, therefore, there are no obvious backtracking points! With this extreme situation a back track is forced by updating $confset(e_i)$ with e_{i-1} as the backtracking point, before the **retreat** function is called.

Let us revisit the backtracking refinement scenario from Figure 3.2, repeated in Figure 3.3. Refinement must backtrack because $Filtered_5 = \{\}$, although $Generated_5 = \{K_{51}, K_{52}, K_{53}, K_{54}\}$. Thus for each KBS in $Generated_5$, at least one of the constraint examples in cebuf must be wrongly answered; suppose K_{51}, K_{52} wrongly solve e_2, and K_{53}, K_{54} wrongly solve e_3. For BT, e_5's confset is the previous refinement example $\{e_4\}$, and refinement proceeds by backtracking to e_4 on the refinement path and choosing the next branch, in this case K_{42} with e_5. For BJ and CBJ, e_5's confset contains the failed constraint examples $\{e_2, e_3\}$. So refinement continues from the most recent on the path, e_3, selecting the next available refined KBS K_{32}, with e_4 as the next refinement example.

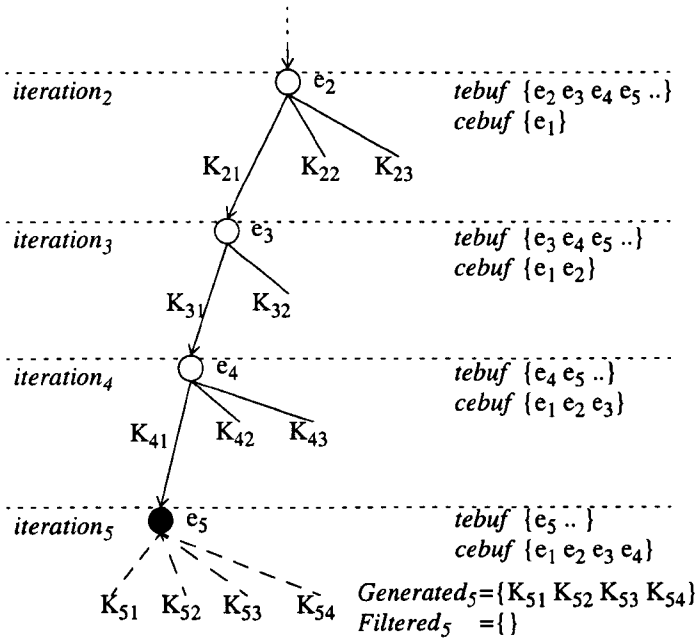


Figure 3.3: Informed backtracking with the KRUSTtool.

The refinement example e_5 is moved back into tebuf as a future refinement example. If no more KBSs are available from e_3 then BJ backtracks to the e_2 node and CBJ backjumps according to the union of e_3's and e_5's conflict sets. Now, consider the situation where $Generated_i = \{\}$, here we have no refined KBSs to analyse inconsistencies with cebuf. An update of e_5's conflict set is forced by adding e_4 as a backtracking point. Consequently, refinement proceeds with backtracking to e_4 choosing to continue refinement with K_{42} with e_5 as the next refinement example.

It is difficult to see how the conflict set contents might be exploited for explanation-based learning with refinement search. Learning might have been possible if there had been additional information that specified the cause and effects of various refinement operators. Work done on knowledge acquisition scripts by Tallis & Gil (1999), provides useful insight in this direction, as scripts not only specify a sequence of refinement steps but also the effects of these steps. However, initial results indicate that the complex interaction between faults in a KBS makes it difficult to enumerate all possible preconditions and effects without making the scripts too specialised or too generalised. Furthermore, if all effects can be identified for a given sequence of refinement steps, it is likely that backtracking will not be required.

3.2.3 Implementation Issues

A certain amount of book keeping during refinement search is essential to enable backtracking to earlier points and restarting refinement with previously abandoned refined KBSs. The data structure that maintains these details are shown in Figure 3.4. This is a typical two-way linked list, with each element in the list linked to the preceding and succeeding elements. Each element itself contains information about the refinement example, the output refined KBS, and the set of refined KBSs maintained as a priority queue, are sorted by accuracy on cebuf and then on tebuf. Any constraint examples that are inconsistent with the refined KBSs are sorted by recency and maintained in the confset. Presently, the complete rule sets of refined KBSs are maintained in each list element as

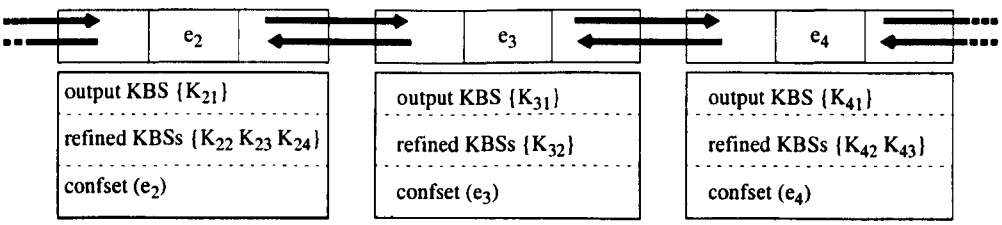


Figure 3.4: Maintaining refinement information.

opposed to maintaining just the refinement changes. The disadvantage of the latter is that it requires merging of refinements and undoing refinements whenever backtracking is triggered, thereby increasing processing time. However, for large KBSs with over 150 or more rules it might be sensible to maintain just the refinement changes, particularly, if storage resources become insufficient.

3.3 Adapting the Informed Backtracking Algorithm

With the analogy in Table 3.2 we are able to employ CSP search techniques for refinement search. However, there are two problems not seen with CSPs that must be dealt with in knowledge refinement:

- the behaviour of constraint examples can change – from being correctly solved and not requiring refinement, to providing new fault evidence in response to a future refinement choice; and

- constraint examples that did not trigger refinement because they were already correctly answered by the input KBS, might appear in conflict sets as potential backtrack points.

To deal with these problems that are unique to refinement search, we look at how the informed backtracking algorithm might be further adapted for refinement search.

3.3.1 Latent to Active Examples

Figure 3.5(a) illustrates, complications that may arise from the changing behaviour of constraint examples. In $iteration_3$ the input KBS K_{21}, already answers e_3 correctly and

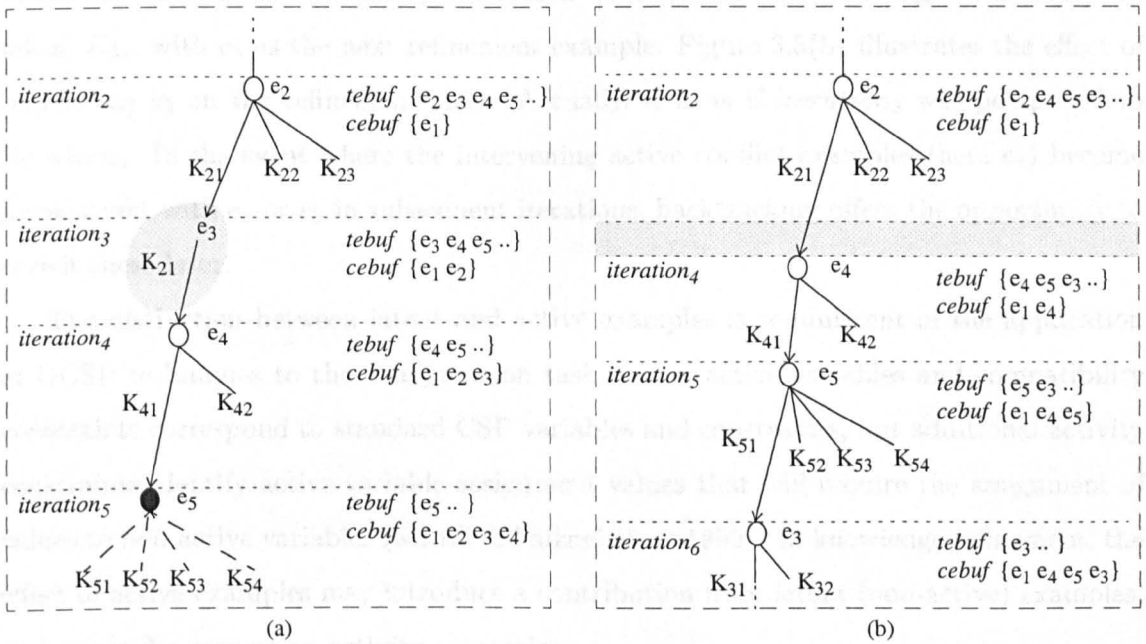


Figure 3.5: Changing behaviour of constraint examples. (a) Refinement Path with a Latent Example. (b) Re-ordering the Latent Example.

so the output from this iteration is the same input KBS, K_{21}; this has been highlighted with grey shading. It does not affect the search when it is advancing, but backtracking to this point will cause problems. Let us assume that in Figure 3.5, backtracking is triggered because $Filtered_5 = \{\}$. Suppose we are using BJ and $confset(e_5)$ is $\{e_2, e_3\}$, so we backjump to e_3, the most recently instantiated conflicting point. But the input KBS K_{21} already correctly answers e_3 and so alternative refined KBSs are not available. We could simply backtrack further, but the KRUSTtool has just discovered a relationship: the changes to correct e_5 have interacted with the way that e_3 was previously solved.

Thus, if we backtrack above e_3 then it is possible that the same interaction will occur again. Instead, we can take this opportunity to advance refinement by exploiting the new relationship that has just been discovered.

Constraint examples that did not contribute fault evidence, like e_3 are called *latent* examples while the other refinement examples are *active*. The activation of latent examples is often due to a fault being exposed as a result of fixing one or more other faults. However the activation may also be due to a previous fix that had incorrectly introduced a new fault. Given the interacting relationship between the latent example e_3 and the active refinement example e_5, we choose to solve their conflict at this point by re-instating e_3 at the top of tebuf and advancing the search with refined KBS with highest accuracy on tebuf, K_{51}, with e_3 as the next refinement example. Figure 3.5(b) illustrates the effect of reinstating e_3 on the refinement path. Actually, it is as if $iteration_3$ was postponed to $iteration_6$. In the event where the intervening active conflict examples (here e_4) become inconsistent with e_3 or e_5 in subsequent iterations, backtracking offers the opportunity to revisit these later.

The distinction between latent and active examples is reminiscent of the application of DCSP techniques to the configuration task, where active variables and compatibility constraints correspond to standard CSP variables and constraints, but additional activity constraints identify active variable assignment values that will require the assignment of values to non-active variables (Mittal & Falkenhainer 1990). In knowledge refinement, the effect of active examples may introduce a contribution from latent (non-active) examples, and so can be seen as an activity constraint.

3.3.2 Prioritising Latent Over Active

The presence of latent examples in a refinement path has no impact initially as they are already answered correctly, and do not provide fault evidence. When latent examples crop up in subsequent confsets, not only do they provide fault evidence, but they also have the added interacting relationship with the current refinement example. Therefore, the backjumping algorithms are amended to take further account of latent examples when they appear in confsets as potential backtracking points. If in Figure 3.5(a), $confset(e_5)$ is $\{e_3, e_4\}$, then backjumping will resume with e_4 and the fault evidence now presented by the latent example e_3 will be lost. Instead, we priorities latent examples that appear in conflict

sets, and rather than backtracking to the most recent conflicting example, we reinstate all conflicting latent examples into the tebuf. This would mean that search proceeds with e_3 and K_{51}, the refined KBS in $Generated_5$ with the highest accuracy, despite e_4 being in the confset. If the intervening active conflict example e_4, still remains a problem, again backtracking offers the opportunity to investigate there later.

3.4 Comparison of Backtracking Search for Refinement

BT suffers from thrashing; rediscovering the same inconsistencies and same partial successes during search. Backjumping schemes reduce BT's unfortunate tendency to rediscover the same dead-ends by retreating search to the actual cause of the inconsistency. We would expect the same situation within knowledge refinement; where backtracking one refinement cycle at a time (BT) is likely to lead to many iterations, so the motivation for introducing BJ and CBJ is to reduce refinement cycles. The experiments apply the Clips KRUSTtool to a corrupted version of the student loans KBS (see Appendix A).

3.4.1 Experimental Design

The training examples had to be carefully selected to ensure that backtracking was exercised, since it is only triggered when conflicting repairs are attempted with interacting faults. A controlled formation of training sets was necessary, as a purely random formation of training sets might not contain examples that expose interacting faults.

The faulty KBS was executed on training sets with varying example sequences, paying attention to the KRUSTtool's refinement path, particularly when backtracking is triggered. When a backtrack or a backjump is triggered with CBJ at node e_j, the confset contents, $confset(e_j)$ are noted. These contents are then used to identify example pairs that have the potential to trigger backtracking whenever included together as part of the training set. Let us assume that $confset(e_j) = \{e_g, e_h, e_i\}$, then the following conflict pairs are noted $\{(e_g\ e_j), (e_h\ e_j), (e_i\ e_j)\}$. Notice that although backtracking happens from one example to another, the cause for backtracking need not always be restricted to a pair of examples. Instead, it could be due to a sequence of incremental refinements associated with 2 or more examples on the refinement path; for instance $\{(e_g\ e_h\ e_j), (e_i\ e_j), (e_h\ e_i\ e_j)$ etc.$\}$. However, it is difficult to meticulously identify all such potential combinations,

instead we opt for the more straightforward formation of conflict pairs.

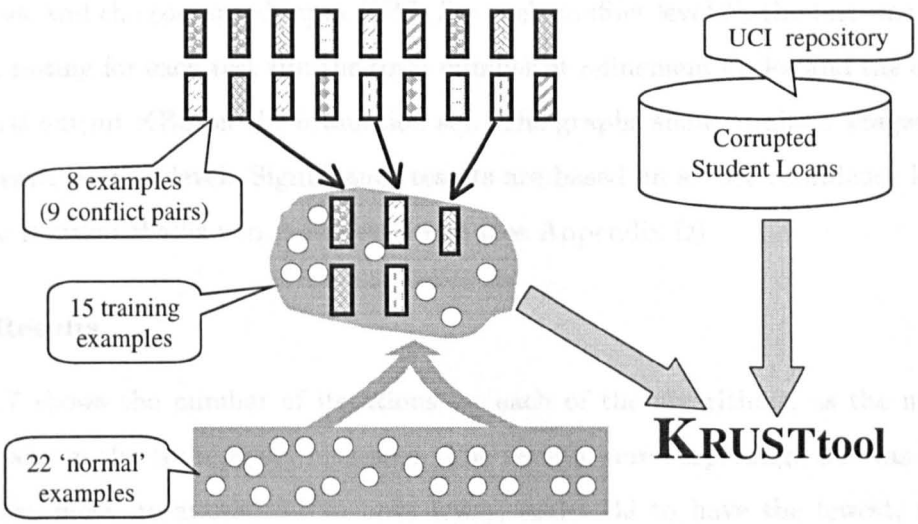


Figure 3.6: Forming training sets that trigger backtracking.

From a selected dataset of 30 examples, 9 conflicting pairs were identified. Actually, the 9 conflict pairs were formed from 8 carefully chosen examples. Therefore, of the 8 examples some are part of 1 or more conflict pairs. Finding conflicting examples was relatively easy given the density of corruption of the KBS. The rest of the 22 examples from the dataset although considered *normal*, cannot be ruled out as containing other conflict pairs. Training sets of a given conflict level N, were created from the selected dataset of 30 examples, by randomly choosing N conflict pairs from the identified 9 conflict pairs, removing duplicate examples when they occurred, and randomly selecting from the *normal* examples, until the training set contained 15 examples.

Figure 3.6 illustrates the formation of a training set, with a conflict level of 3. Each of the 9 pairs of shaded boxes (one stacked on top of the other) denotes a conflict pair. There are 8 types of shading representative of the 8 examples that form the 9 conflict pairs. Once the 3 conflict pairs are randomly selected, any duplicates are removed, hence the single shaded box amidst the other 2 selected pairs. A training set of 15 examples are formed by randomly selecting a further 10 examples from the *normal* set. The unselected examples from the dataset form the evaluation set. The training set is KRUSTtool's tebuf and the faulty KBS is refined based on fault evidence generated by examples in tebuf. Refinement continues until all examples in tebuf are correctly solved.

The KRUSTtool incorporating the BT, BJ and CBJ algorithms were applied to each training set and the corrupted input KBS. For each conflict level N, the test was repeated 10 times, noting for each test run the total number of refinement cycles and the error-rate of the final output KBS on the evaluation set. The graphs show results averaged over 10 runs for each conflict level. Significance results are based on a 95% confidence level, and apply the Kruskal Wallis non-parametric test (see Appendix D).

3.4.2 Results

Figure 3.7 shows the number of iterations for each of the algorithms, as the number of conflict pairs in the training set increases. The results were surprising. BT was expected to have the most iterations, BJ to have fewer, and CBJ to have the fewest, reflecting the increased targeting of the search. Instead, we see that BJ has utilised a significantly greater number of iterations (p=0.001). The increased iterations with conflict pairs 3, 5 and 9 is explained by the random selection of conflict pairs during training set formation. Essentially, when several conflict pairs are selected without any overlap there will be fewer *normal* examples needed to form the training set. Such a training set will naturally be more demanding on the refinement process. With CSPs, BT is guaranteed to have at least as many iterations as BJ or CBJ. However, in the more dynamic space of refined KBSs this is not the case; backjumping searched a different part of the space that involved more iterations.

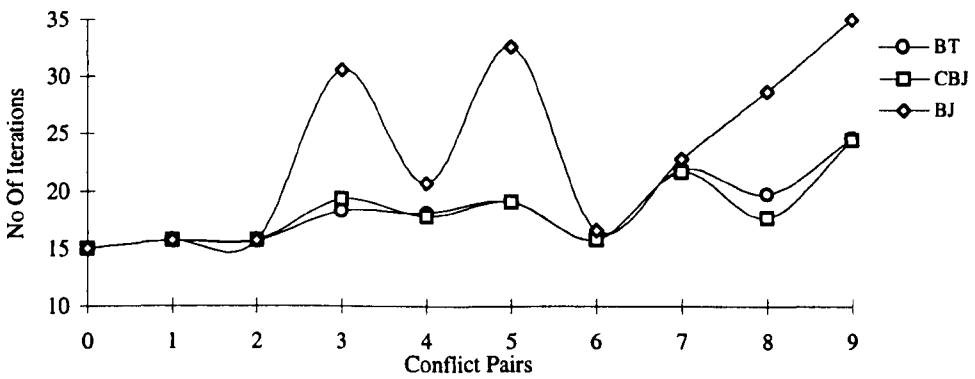


Figure 3.7: Number of iterations (Basic Algorithms).

So has there been any gain from BJ's additional searching? Figure 3.8 shows the error rates of the final KBS produced by the 3 algorithms on the complete set of 30 examples;

the error-rate of the original corrupted KBS is the horizontal dashed line on all error-rate graphs. BJ, the most greedy in refinement cycles, has indeed gained the lowest error rate (p=0.005). This behaviour is explained by noticing that, although BJ and CBJ are guaranteed to find all binary CSP solutions, this is not the case with refinement, since refinements in different cycles can interact: an earlier refinement can provide part of a later refinement or conflict with the later refinement. Therefore the refinements that are proposed depend on the input KBS, and thus the refinement path.

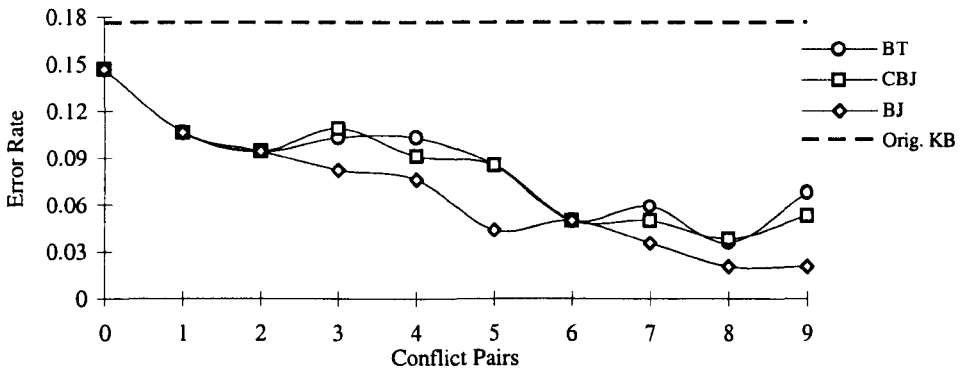


Figure 3.8: Error rate of final refined KBS (Basic Algorithms).

3.5 Exploiting Conflict Knowledge

Figure 3.8 shows another interesting trend: the error rate of the refined KBS decreases as the number of conflict pairs in the training set increases. This confirms the experimental results in (Palmer & Craw 1996), that the more demanding the examples in the training set the higher the accuracy achieved by refinement. It also suggested that we explore re-ordering the training examples to exploit conflict pairs as soon as it is recognised during iterative refinement. The Minimal Bandwidth Ordering heuristic for static ordering of variables, attempts to reduce the backtracking distance for CSP algorithms, by placing mutually constraining variables close together in the search (Tsang 1993). The previous Section recognised that the refinement example, e_j, and its conflicting examples, *conf-set*(e_j) are mutually constraining, since refinements for e_j had affected the correctness of previous latent examples. We use this idea of mutually constraining examples, to associate each refinement example and its deepest conflicting constraint example in the sequence

of training examples, in an attempt to reduce the number of iterations of the informed backjumping algorithms, without compromising the error-rate of the final output KBS.

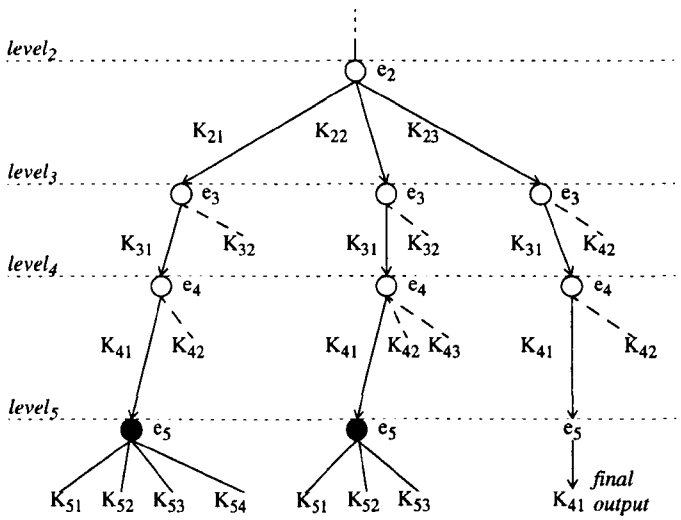


Figure 3.9: Searching without Conflict-Based re-ordering.

Figure 3.9 illustrates a hypothetical backjumping situation. The refinement search space contains three main refinement paths, of which two have been discarded: $e_2 \xrightarrow{K_{21}} e_3 \xrightarrow{K_{31}} e_4 \xrightarrow{K_{41}} e_5.\emptyset$ and $e_2 \xrightarrow{K_{22}} e_3 \xrightarrow{K_{31}} e_4 \xrightarrow{K_{41}} e_5.\emptyset$. Suppose in each situation $confset(e_5)=\{e_2\}$ and so backjumping to e_2 produces the search as illustrated. But this also means that e_2 and e_5 are mutually constraining since the repairs to e_5 has affected the solution to e_2. The Minimal-BJ (MBJ) and Minimal-CBJ (MCBJ) algorithms contain a further amendment to the informed backjumping algorithms, so that backjumping to a node e_i that conflicts with the current refinement example e_j, causes the algorithm to try to fix this pair of mutually constraining examples next. It re-sorts tebuf so that e_j is re-used with the next refined KBS from e_i. Thus, the pair of conflicting examples identified in backjumping become adjacent on the new branch of the refinement path. Figure 3.10 illustrates the outcome of re-ordering tebuf examples so that e_5 is used as the next refinement example after backjumping to e_2, and indicates the potential saving in iterations over Figure 3.9. Although this re-ordering is not guaranteed to reduce iterations, the relationship between an example and its confset gives some justification for re-ordering the otherwise random order of the training examples. It is possible that successive re-ordering of nodes in this manner may at times lead to the original sequence. Even so, this will not result in cycling because BJ and CBJ will resort to backtracking once all branches of a node are explored.

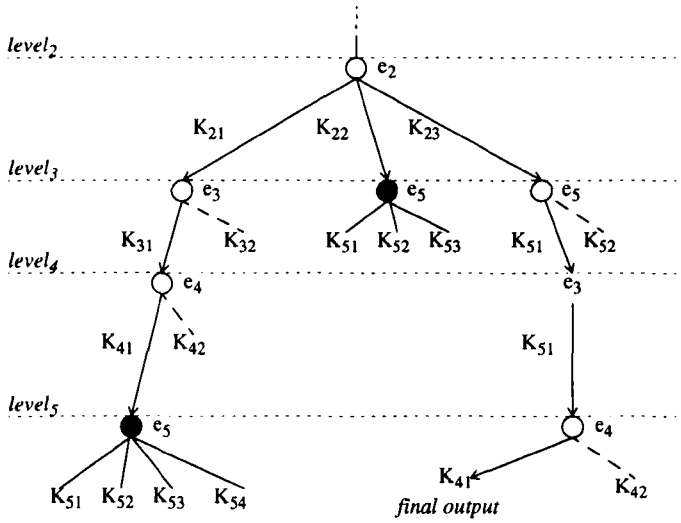


Figure 3.10: Searching with Conflict-Based re-ordering.

The earlier experiments were repeated with MBJ and MCBJ algorithms. Figure 3.11 superimposes the bar chart for MBJ iterations on the line graphs for the basic algorithms; the results for MCBJ are similar to CBJ's so are not shown on the graph. Our goal of reducing the number of iterations in BJ has been achieved in general, and MBJ's iterations are closer to BT and CBJ. There were 3 test runs where BJ performed fewer iterations than MBJ, and a closer examination of one indicated that re-ordering resulted in an increased search space when two examples e_i and e_j are affected by the same repair, where the fault exposed by e_j cannot be correctly refined before the fault exposed by e_i is refined. Dependencies of this nature suggest the existence of refinement interdependencies between training examples, and we explore heuristics that might help identify such relationships in Chapter 5.

Figure 3.12 confirms that the refined KBS error rates with MBJ, and CMBJ, are unaffected by the dynamic re-ordering. So MBJ has achieved fewer iterations without increasing the error-rate of the final KBS.

3.6 Conclusion

We have transformed the natural hill-climbing of the KRUSTtool refinement algorithm into a best first search with the potential to revisit previously discarded refined KBSs. It is the KRUSTtool's ability to generate many potential refined KBSs in response to fault

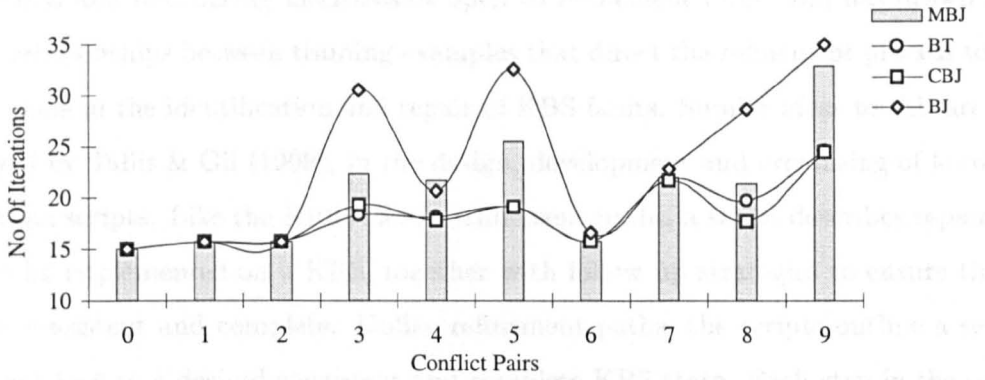


Figure 3.11: Number of iterations (Conflict-Based re-ordering).

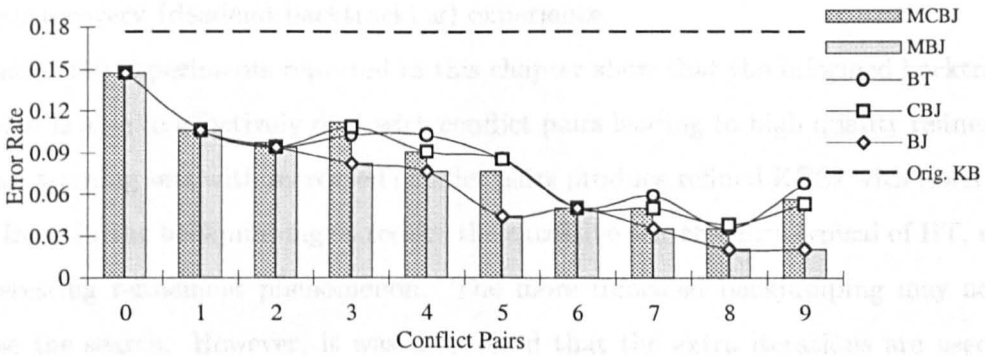


Figure 3.12: Error rate of final refined KBS (Conflict-Based re-ordering).

evidence, that enables CSP search strategies to be applied with the central refinement algorithm. Adapting CSP search techniques within the knowledge refinement framework was possible because, these techniques are sufficiently general and a wide range of tried and tested algorithms are available.

The authors of other refinement algorithms (Ourston & Mooney 1994, Richards & Mooney 1995) have argued that the choice of repairs available to their refinement tool is sufficiently flexible that hill-climbing problems occur rarely, and so make no explicit attempt to deal with it. Our testbed has shown that it is relatively easy to find mutually conflicting training examples for sufficiently corrupted KBSs. Therefore, it is important that refinement tools that deal with examples one at a time must be equipped to handle conflicting repairs because it is difficult to deal with mutually constraining examples in a single refinement iteration.

The refinement search that is focused by backtracking, highlights the variety of refine-

ment paths and re-ordering mechanisms open to refinement tools, and has drawn attention to relationships between training examples that direct the refinement process towards staged goals in the identification and repair of KBS faults. Similar ideas to this are being employed by Tallis & Gil (1999), in the design, development and organising of knowledge acquisition scripts. Like the KRUSTtool's refinement paths, a script describes repairs that need to be implemented on a KBS, together with follow up strategies to ensure that the KBS is consistent and complete. Unlike refinement paths, the scripts outline a series of steps that lead to a desired consistent and complete KBS state. Each step in the series is tried and tested, and assumed to be the correct refinement decision, therefore, backtracking steps are not needed. Essentially, scripts can be viewed as refinement paths without the error-recovery (deadend-backtracking) experience.

Finally, the experiments reported in this chapter show that the informed backtracking algorithm is able to effectively deal with conflict pairs leading to high quality refinements, and that training sets with increased conflict pairs produce refined KBSs with lower error-rates. Introducing backjumping to reduce the excessive search effort typical of BT, reveals an interesting refinement phenomenon. The more informed backjumping may actually increase the search. However, it was discovered that the extra iterations are used profitably and provides refined KBSs with lower error-rates. Amendments to the backjumping algorithms to reduce the iterations, whilst maintaining the high accuracy, concentrate on re-ordering training examples once backtracking is initiated, by recognising the information gain offered by both latent and active examples in the confset. These initial experiences with re-ordering, provides the impetus for the next Chapter, where we will investigate elaborate re-ordering techniques that exploit knowledge about refinement generation.

Chapter 4

Refinement Search Efficiency

BJ was introduced as a way to reduce the search effort of BT. Contrary to expectation we found that BJ often increases the number of refinement cycles but that these extra cycles were used profitably. In this Chapter, we are interested in improving search efficiency of the BJ KRUSTtool, by reducing the number of refinement cycles whilst maintaining the improved accuracy as reported in Section 3.4. For this purpose the proposed analogy between CSPs and iterative knowledge refinement is taken a step further with the aim of incorporating CSP ordering strategies for improving search efficiency within the iterative refinement framework. The results reported in this chapter have also been published in (Wiratunga & Craw 1999b).

Section 4.1 discusses constrainedness with respect to iterative refinement and introduces three heuristics that can be employed to reduce refinement search effort. Experiments comparing various ordering heuristics on the Student loans domain are presented in Section 4.2 followed by Chapter conclusions in 4.3.

4.1 Constrainedness of Refinement Search

Value ordering in CSPs is analogous to ordering of refined KBSs; which is already done by the KRUSTtool by means of the accuracy ranking. In fact KRUSTtool's informed backtracking algorithm is closely related to the repair-based approach to solving CSPs and its greedy min-conflict heuristic for repair selection (Minton et al. 1992); and the refined KBS ordering itself is similar to the look-ahead value ordering min-conflicts heuristic that ranks the values of a variable in increasing order based on the number of incompatibilities with

values of future variables to be instantiated (Frost & Dechter 1995). Here, we concentrate on how variable ordering can be applied to iterative knowledge refinement.

Although, the notion of constrainedness of variables for CSPs is straightforward, how does constrainedness translate to training examples for iterative knowledge refinement? CSP variables involved in the most or tightest constraints correspond to training examples where refinements generated by the KRUSTtool puts the highest consistency demands on previously solved training examples. For instance, we can think of the number of constraints example e is involved in, as the number of previously solved examples that get unsolved when attempting to solve e. We investigate how these mutually constraining examples might be identified and the effect of dealing with them first in the next Chapter.

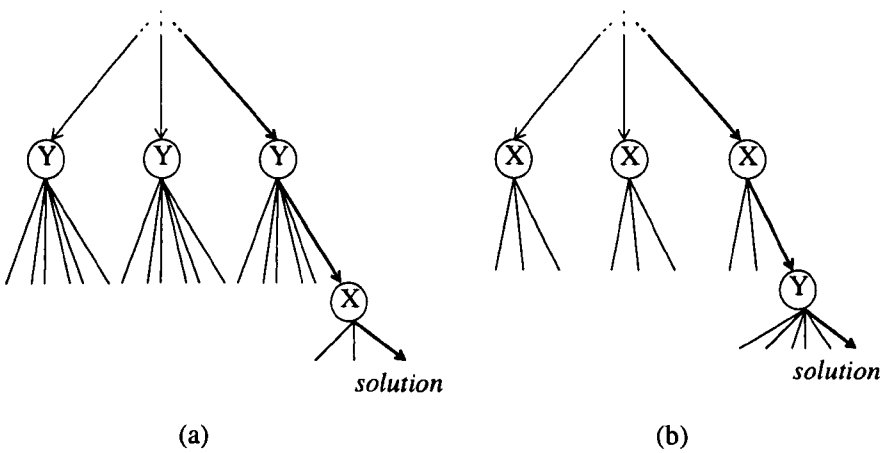


Figure 4.1: Search space (a) without ordering and (b) with ordering

CSP variables with smaller domains correspond to refinement examples that resulted in the generation of fewer refined KBSs in a refinement cycle. In this chapter we enforce example ordering according to the number of generated refined KBSs. This means that the KRUSTtool will deal with refinement examples that are likely to provide fault evidence resulting in the generation of fewer refined KBSs first. Accordingly, *constrainedness* of a refinement example can be defined as the number of potential refined KBSs that are generated by the KRUSTtool in response to fault evidence. The fewer potential refined KBSs the fewer refinement paths, and the more constrained the refinement example. Therefore, it is in KRUSTtool's best interest to deal with most constrained examples first, as early failure helps prune the search space, thereby reducing overall search effort. In CSP literature this strategy is referred to as the "fail-first principle"(FFP) (Haralick & Elliott 1980).

Figure 4.1 illustrates the motivation behind the FFP. Here, refinement examples X and Y are both constrained by previously solved refinement examples and are likely to trigger backtracking. Obviously, the difference in the number of generated refined KBSs with each of the refinement examples (6 with Y, and 3 with X), means that the order in which these refinement examples are dealt with can have a marked difference on the search effort. In Figure 4.1(b), by first attempting X, the example with fewer refined KBSs, the KRUSTtool is more likely to encounter dead-ends early on. Essentially, if we know how many potential refined KBSs might get generated with each example, we are able to enforce some order on the training examples. Certainly, going as far as refinement generation to measure constrainedness of training examples can be computationally expensive. Instead we establish heuristics that *estimate* constrainedness for each training example and use these estimates to enforce an order on the training examples.

4.1.1 Evidence From the Recent Refinement Cycle

Simple constrainedness information comes from the newly completed refinement cycle; where the final step executed all the refined KBSs generated in that cycle on the remaining training examples in tebuf. Although this was done to calculate the error-rate of each of these refined KBSs, it also determines an estimate of how faulty each training example is; i.e. how many of these refined KBSs solved the training example incorrectly. With increased numbers of refined KBSs failing to correctly solve an example, the greater the evidence that the example is constrained. Remember that all these refined KBSs are related since they were all derived from the same input KBS, therefore, this justifies employing them to select the next most constrained refinement example. The underlying intuition is that an example from tebuf for which the generated refined KBSs find difficult to solve will have the greatest consistency demands and be restricted to fewer number of refined KBSs being generated in the subsequent refinement cycle.

Let us assume that M refined KBSs $K_{i1}, K_{i2}, \ldots, K_{iM}$ were generated with e_i as the refinement example and that tebuf now contains training examples e_{i+1}, e_{i+2},...,e_n. Table 4.1 demonstrates how fault evidence from the most recent refinement cycle can be employed to select the next refinement example from tebuf, with $M = 3$ and $n = 4$. The table entry for e_j and K_{ik} has value 1 if K_{ik} incorrectly solves e_j, and 0 otherwise. Therefore, the *error-rate* of K_{ik} on tebuf, $err_{K_{ik}}$, is the column total divided by n. The

row total $fault_j$ is the level of *faultiness* of e_j as judged by $K_{i1}, K_{i2}, \ldots, K_{iM}$. The refined KBS with the lowest error-rate, $\min(err_{K_{ik}})$, is selected as the best refined KBS. For ordering purposes, we use the faultiness measure, where the training example with the highest level of faultiness, $\max(fault_j)$, is selected as the next refinement example. In Table 4.1 K_{11} with lowest error rate is selected as the best refined KBS, while e_2 with maximum faultiness is selected as the next refinement example.

Generated Refined KBSs

tebuf		K_{11}	K_{12}	K_{13}	*faultiness*
	e_2	1	1	1	3
	e_3	0	1	0	1
	e_4	0	1	1	2
	e_5	0	1	1	2
	error-rate	0.25	1	0.75	

Table 4.1: Constrainedness of training examples using potential refined KBSs.

This heuristic is reminiscent of the best known CSP dynamic ordering heuristic, dynamic search rearrangement (DSR), which selects the next variable having the minimal number of values that are consistent with the current partial solution (Dechter & Meiri 1994). Heuristically, the choice of such a variable minimizes the remaining search. With knowledge refinement we use fault evidence about the most recent potential refined KBSs as the basis for selecting the most constrained training example for the next iteration.

4.1.2 Evidence From How the Problem was Solved

A more direct estimate of how many refined KBSs will be generated for a particular training example is the number of places where the problem solving behaviour for that training example can be changed. The KRUSTtool algorithm creates a data structure containing precisely this information. The *problem graph* captures the problem-solving for the refinement example and allows the KRUSTtool to reason about the fault that is being demonstrated (Craw & Boswell 1999). Essentially, the problem graph records the sequence of rule activations leading to the system solution, and additionally shows all possible rule activation routes that could lead to the required goal solution. Problem graphs can become quite complex with long chains and complicated branching.

In figure 4.2 we use a fictitious rule base, sufficient to illustrate three simple problem graphs and their function. With training example A=$\langle[f_{A1},\ldots,f_{A4}],goal_A\rangle$, the KBS currently reasons from the observables by applying leaf rules R_7 and R_4, which together allow a middle rule R_{13} to fire, and finally the end rule R_{11} concludes S_A, a faulty system solution. The darkened area of the problem graph is the *positive problem graph* and corresponds to the problem solving that has been undertaken by the faulty KBS. Therefore it contains the solution subgraph for the training example but also contains other partial proofs; e.g. f_{A1} allows R_7 to fire, but this only partially satisfies R_{12}. The positive problem graphs for the other two training examples are similar but notice neither provides a system solution since each partial solution subgraph terminates with an intermediate result.

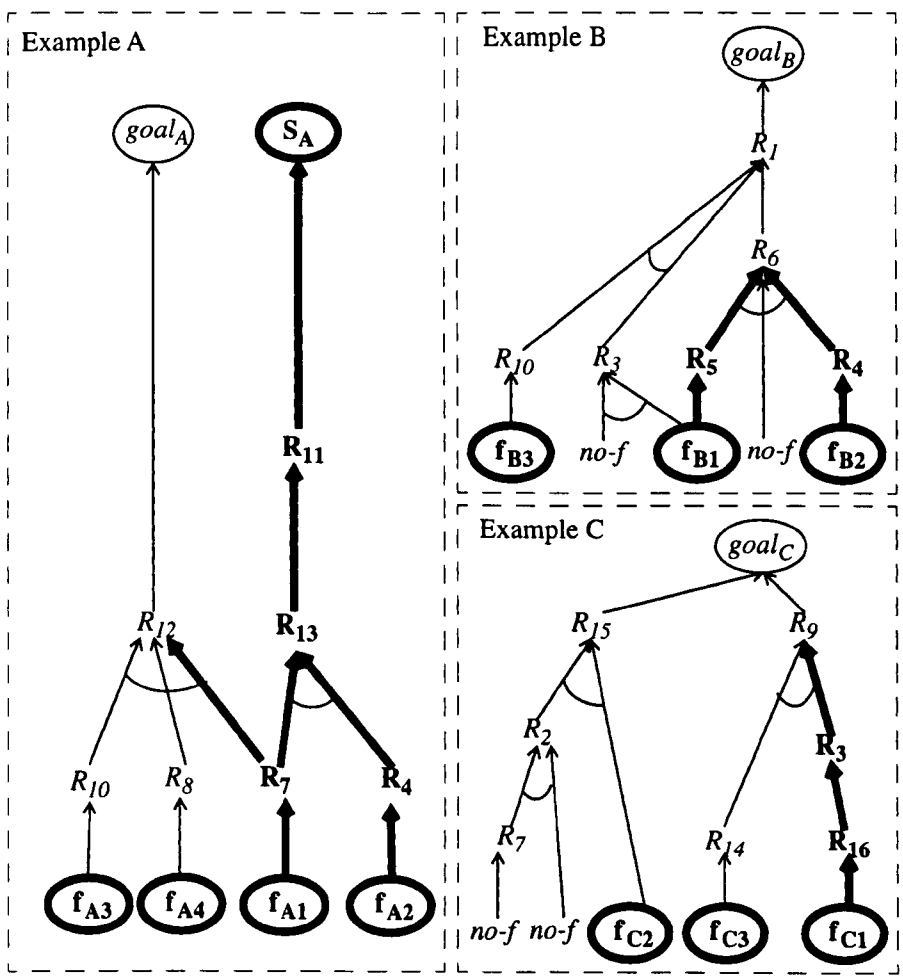


Figure 4.2: Problem Graph for training examples, A, B and C.

Repairs correspond to preventing faulty rule chains from being activated and so the

number of rule activations in the positive problem graph is a simple measure to predict the number of potential refined KBSs, and hence how constrained the refinement cycle for that training example will be. Essentially, with fewer number of rule activations the more constrained the refinement cycle, because there are fewer refinement points. According to the problem graphs in Figure 4.2 training examples A, B and C have activation counts 4, 2 and 2 respectively, indicating that B and C are the most (and equally) constrained and so will be selected before A. All ties are broken randomly.

A different yet interesting implication of ordering according to rule activation measures is that typically, fewer rule activations suggests a faulty KBS that is too specialised. Training examples with fewer activations are most likely to be false negatives and will be dealt with first. This is analogous to the ordering strategy adopted by AUDREYII (see Section 2.1.2), where false negatives are processed before false positives. The difference is that with a faulty KBS that is too generalised, counting rule activations provides us additional information about constrainedness, while AUDREYII would simply deal with false positives in no particular order.

4.1.3 Evidence From How the Problem Should be Solved

The problem graph captures more about the problem-solving than simply recording what happened. It also contains the *negative problem graph* which models all possible rule non-activation routes that would lead to the goal. These are the rule activations that did not happen, and if they did would have resulted in the required goal, i.e. the expert's solution. Therefore, in Figure 4.2 the expert's solution for training example A ($goal_A$) has not been proved because, R_{10}, R_8 and R_{12} are only partially satisfied, and are unable to activate. The arrows leading from f_{A3} and f_{A4} have not been darkened to indicate that the conditions in R_{10} and R_8 do not match observables f_{A3} and f_{A4}, and must be weakened (generalised) before they are satisfied. In contrast conditions in R_4, R_{13} or R_{11} must be strengthened in order to stop the incorrect system solution, S_A being asserted. Similar explanations hold for training examples B and C, but now in addition some rule conditions (e.g. the first condition of R_3 for training example B), cannot be weakened to match any observable or rule conclusion and so are not linked to any rule or observable but instead these "non-observables" are labelled *no-f*.

The negative problem graph provides additional information on how constrained the

refinement cycle will be. Counting all the rule activations from the positive parts and the non-activations from the negative parts of the problem graph provides a second measure of constrainedness. This measure promises to be more informative since it adds the locations of possible repairs in the negative problem graph to those from the positive part.

In practice it makes sense to distinguish between rules in the negative problem graph whose conditions could be weakened to match observables from those that could never be matched. For this purpose the heuristic is fine tuned so that it ignores any negative rule activation whose conditions are *all* linked to (or derived from) "non-observables" (*no-f*'s in the diagram); e.g. rules R_7 and R_2 will be omitted from C's count. Without this modification the heuristic can estimate a training example like C to be less constrained than it actually is. Such an amendment requires the assumption that training examples are noise free, however this seems acceptable given our need simply to estimate constrainedness.

Training	*Refinements*				*Rule Activations*	
Example	*Strengthen*	*Weaken*	*None*	*Count*	*All*	*Improved*
	R_4	R_{10}	R_7	6	7	7
A	R_{13}	R_8				
	R_{11}	R_{12}				
		R_1	R_5	4	6	6
B		R_3	R_4			
		R_{10}				
		R_6				
		R_9	R_3	3	7	5
C		R_{14}	R_{16}			
		R_{15}	R_7			
			R_2			

Table 4.2: Refinements and rule activations from the complete problem graph.

Table 4.2 lists all the refinement points for the three training examples A, B and C, at the left. The count of rule activations in the complete problem graph, with and without the non-observables correction, appears at the right. Therefore, example C with the lowest improved rule activation count is selected over A and B. Notice that although the improved heuristic is a good predictor of the number of refinements here, more complex problem graphs may need a more sophisticated measure that takes into consideration the overlap between the positive and negative graphs as this may suggest an area void of refinement points.

4.2 Ordering Heuristics in Practice

We can now evaluate the BJ KRUSTtool by applying static and/or dynamic ordering of the training examples using the heuristics described in Section 4.1. Static ordering involves an initial ordering of training examples, while dynamic ordering ensures that the order of training examples for a given refinement cycle is influenced by the most recent refinement cycle. The problem graph heuristics define a static ordering of the training examples before the iterative refinement cycles are started. They can also be used for dynamic ordering where the measures are recalculated on the best refined output KBS from a refinement cycle and applied to re-order the remaining training examples in tebuf. The fault based heuristic (Section 4.1.1) can only be applied as a dynamic ordering since it exploits information from all generated refined KBSs from the previous refinement cycle. We expect ordering to reduce the search effort thereby reducing the number of refinement iterations. Although the emphasis of the evaluation is to compare the number of iterations, it is important to establish the effect of ordering on the error-rate. Additionally, it is necessary that any substantial increase in cpu usage be justified by reduced search effort.

The testbed for the experiments in this Chapter use the corrupted Student loans KBS (Appendix A). We re-use the 20 training/test splits corresponding to conflict levels 5 and 9 from Figure 3.7, since the BJ KRUSTtool was shown to have the greatest number of iterations with these sets. Moreover, as the assessment of BJ efficiency with various orderings of training examples require that backtracking is triggered, it makes sense to re-use these training sets containing conflict pairs which were formed with backtracking in mind.

4.2.1 Static Ordering

Static ordering provides a sequence of training examples prior to the iterative refinement cycles. We compare two orderings using the problem graph heuristics with a random ordering.

- NOORDER: move all correctly solved training examples into cebuf then randomly order tebuf.
- PGRAPH+: move all correctly solved training examples into cebuf, then sort the

remaining training examples in decreasing order of the number of rule activations in the positive problem graph only.

- PGRAPH±: as PGRAPH+ but use the number of rule activations in the complete problem graph (positive and negative) including the modification for "non-observables".

Static ordering	*Mean*	*Median*	95% *Confidence*
NOORDER	9.05	8.0	±1.420
PGRAPH+	7.65	7.0	±0.717
PGRAPH±	7.65	7.5	±0.410

Table 4.3: Number of iterations for static ordering.

Error-rate for the final refined KBS was not impaired with PGRAPH+ and PGRAPH±, and they both reduced the error-rate compared to NOORDER in 4 test runs. More pertinent to this evaluation is the number of iterations for these three algorithms listed in Table 4.3. PGRAPH+ required significantly fewer iterations compared to NOORDER (p-value = 0.028); 10 test runs had fewer iterations and only 2 test runs had more iterations and this was at most 2 iterations longer. The reason for these extra iterations in 2 test runs is explained by the dynamic nature of iterative refinement, where the estimated *domain* (refined KBSs) of refinement examples can change with incremental refinement. However, such problems can only be tackled by re-ordering after each refinement cycle as a single static ordering right at the start is not sufficient.

PGRAPH± improved on PGRAPH+ by reducing the number of iterations in 4 test runs, however despite the added information acquired from the negative problem graph this reduction is not statistically significant. Any improvements in PGRAPH± over PGRAPH+ is due to the added information causing fewer ties, which essentially means fewer randomly resolved tie-breaks. This surprisingly (only) marginal improvement of PGRAPH± over PGRAPH+ is explained by observing that refinement generation explores both the positive and negative problem graphs and that refinements can include changes to both parts of the reasoning. Therefore a more complex combination of rule activation counts may be required so that it takes account of those activations that contribute towards the required goal and are also part of the positive problem graph, by not counting them as individual activations.

The reason for the reduced number of iterations with NoOrder when compared to the values reported in Figure 3.7 for the same test runs is explained by the moving of all correctly solved examples into cebuf right at the start. Overall, the test results clearly indicate that the order in which training examples are processed by the KRUSTtool affects the number of backjumps and iterations. It also confirms that the number of rule activations is an indicator of the level of constrainedness of a training example.

4.2.2 Dynamic Ordering

The original backjumping KRUSTtool already employs two forms of dynamic ordering.

- Reinstating constraint examples that did not require refinement at the time, these are latent examples that did not provide any fault evidence as refinement examples, but are now incorrectly solved by the current KBS and so are moved back into tebuf (see Section 3.3).
- Re-ordering of examples with MBJ and MCBJ algorithms, where two mutually constrained examples are dealt with in consecutive refinement cycles (see Section 3.5).

Both these reordering strategies are applicable only when backjumping occurs. We now extend training example ordering by applying each of the three heuristics from Section 4.1 to also reorder before every refinement cycle. This form of general reordering is employed first, to ensure that reordering enforced by backjumping is not undone.

Figure 4.3 outlines the basic algorithm combining static and dynamic ordering for the BJ KRUSTtool algorithm. Any of the three static orderings NoOrder, PGraph+, PGraph± from Section 4.2.1 can be used in step 2 and will only influence the selection of the first refinement example. Dynamic ordering occurs in step 3c, where any of the following can be applied:

- FaultBased: re-order tebuf in decreasing order according to evidence from KBSs from the recent refinement cycle (Section 4.1.1), after moving all correctly solved training examples from tebuf into cebuf; or
- DynPGraph+: apply PGraph+'s heuristic (now in every cycle); or
- DynPGraph±: apply PGraph±'s heuristic (now in every cycle).

1. Current best refined KBS is the input faulty KBS.
2. Apply static ordering on tebuf.
3. Loop until tebuf is empty:
   (a) Execute the top most example in tebuf on the input KBS.
   (b) Generate and implement refined KBSs.
   (c) Apply dynamic ordering on tebuf.
   (d) If the set of filtered refined KBSs is not empty then choose the current best refined KBS as the output KBS.
   (e) If the set of filtered refined KBSs is empty:
      i. If there are latent examples then these are pushed into tebuf, after all correctly solved training examples are moved into cebuf.
      ii Otherwise, employ BJ to identify the inconsistent example and its next best refined KBS to backtrack to, and all constrain examples on the way are moved back into tebuf.

Figure 4.3: Algorithm combining static and dynamic ordering.

4.2.3 Static and Dynamic Combinations

The experiments in this section investigate seven (of the nine possible) static-dynamic combinations; the same problem graph and faultiness heuristics are used in the static and dynamic orderings. Once again the error-rate of the final KBS was unaffected. Comparing the results in Table 4.4 with the static ordering results in Table 4.3, we see that all combinations have reduced the number of iterations by at least two iterations. All heuristics employing the complete problem graph resulted in lower average number of iterations but FAULTBASED results are very close. However the differences among all the static + dynamic combinations are not substantial; PGRAPH± + DYNPGRAPH± has the fewest iterations but this is not significant ($p = 0.932 > 0.05$). These results show that using static combined with dynamic ordering gives significant gain over using static ordering only but that none of the combinations is better than any other.

We have succeeded in reducing the number of iterations but at what computational cost? Table 4.5 shows the number of cpu cycles for the heuristic combinations; the entries for static ordering have only been included for reference. FAULTBASED has been very effective for dynamic ordering since the overhead of applying it with any static ordering is not significant. The orderings based on problem graphs have not been so effective; any gain in reducing the iterations has been overwhelmed by the expense of each iteration.

Static	+	*Dynamic*	*Mean*	*Median*	95% *Confidence*
NoOrder	+	FaultBased	5.15	5	±0.532
NoOrder	+	DynPGraph+	5.40	5	±0.765
NoOrder	+	DynPGraph±	5.15	5	±0.613
PGraph+	+	FaultBased	5.60	5	±0.864
PGraph+	+	DynPGraph+	5.80	5	±0.893
PGraph±	+	FaultBased	5.10	5	±0.524
PGraph±	+	DynPGraph±	5.05	5	±0.557

Table 4.4: Number of iterations for static + dynamic ordering combinations.

		Static		
		NoOrder	PGraph+	PGraph±
Dynamic	None	286480	453030	384910
	FaultBased	246060	454590	398670
	DynPGraph+	477760	564810	
	DynPGraph±	581020		798910

Table 4.5: CPU cycles for static + dynamic ordering combinations.

4.3 Conclusion

The refinement search space is extremely dynamic with sequences of refinement examples altering the refined KBSs being considered. As with CSP variable ordering, the goal is to reduce search effort by enforcing an order on training examples. Unlike CSPs, where an instantiation for one variable can only restrict the domains of others, in iterative knowledge refinement the repair for one training example may also lead to a totally different set of proposed refinements for later training examples. However, the dynamic ordering of training examples re-orders examples such that changes in the number of refined KBSs is taken into consideration before selecting the next refinement example. Overall, experimental results show that both static and dynamic ordering heuristics are able to reduce search effort without reducing accuracy.

The problem graph related heuristics for static ordering had significantly reduced search effort when compared to NoOrder. Surprisingly, the additional information from using the complete problem graph instead of just the positive problem graph, did not yield any significant benefits. This suggests the need for a more informed measure that considers the overlap between the positive and negative problem graphs for the PGraph± heuristic.

Dynamic ordering was able to significantly reduce the search effort compared to static

ordering. However, an important issue with dynamic ordering is the additional computational effort introduced by the reordering at each refinement cycle. FAULTBASED very effectively guides the search without substantial computation and for one combination actually lowered the total effort when compared to all static ordering results. In contrast the problem graph heuristics are computationally more expensive. This suggests that an initial random ordering coupled with a FAULTBASED dynamic ordering provides the best balance between improved efficiency and computation costs.

Chapters 3 and 4 have concentrated on improving refinement effectiveness and efficiency by informed *use* of training examples. A separate but important issue for knowledge refinement is informed *selection* of training examples. Here the aim is to reduce labelling and processing costs by selecting few yet informative examples for refinement. This forms the focus for Chapters 5 and 6, where selection mechanisms that are suited for knowledge refinement are presented and evaluated. However, it is important to note that research work on knowledge refinement exploiting CSP search techniques does not conclude here, as it has much growth potential and needs to be extended further.

Chapter 5

Informed Selection of Refinement Examples

Fundamental to a KRUSTtool's successful refinement operation is the availability of labelled examples (for its buffers). Availability is often constrained by limited expert interaction and in this chapter we investigate how a KRUSTtool might benefit from active selection techniques that enable the selection of refinement examples from an available set of unlabelled examples. The goal is to select few yet *good* examples, and by this we mean selecting few examples whilst ensuring that they are representative of the faults in the KBS. The results reported in this chapter have also been published in (Wiratunga & Craw 2000).

Section 5.1 describes how sampling can be incorporated within the iterative refinement framework. The selection strategy in Section 5.2 employs clustering of examples where similarity between unlabelled examples is according to the problem-solving behaviour of the KBS. Section 5.3 identifies several strategies for selecting a suitable number of examples from these clusters. Experimental results from evaluating the selection strategies on two problem domains which have different problem-solving characteristics is presented in Section 5.4, followed by chapter conclusions in Section 5.5.

5.1 The Selective Sampling Process

The choice of training examples for refinement becomes important, when one of the constraints on the refinement process is a limited number of labelled training examples. This is

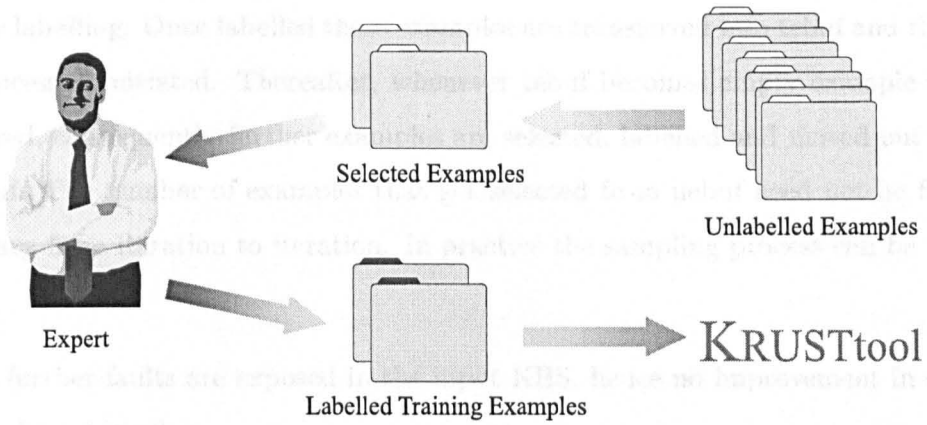


Figure 5.1: A single iteration of *select-label-refine*.

a relatively common problem in a real environment, where labelling many problem-solving tasks with the expert's solution, may require significant interaction with a busy expert. Unlabelled training examples are often generated by using domain knowledge already embodied in the KBS, or meta-knowledge (Zlatareva & Preece 1994, Ayel & Vignollet 1993). Therefore, unlike the labelling task, generating unlabelled examples does not typically require the expert. Here we assume the availability of a sufficiently large set of unlabelled examples. The goal is to perform an informed selection from this set which the expert must label, thereby reducing the demand on the expert. The sampling process consists of 3 stages:

- informed selection of examples;
- labelling of these selected examples by the expert; and
- refinement of the faulty KBS using the batch of newly labelled examples.

This three-stage process of *select-label-refine* is illustrated in Figure 5.1, where a single sampling iteration provides a small batch of labelled examples that can be used by the KRUSTtool. Once the KRUSTtool has incrementally refined the KBS to correctly solve these labelled examples, the next iteration of *select-label-refine* can be triggered. Figure 5.2 illustrates how sampling can be incorporated within the iterative refinement process. The *unlabelled example buffer* (uebuf) contains all unlabelled examples and at the start of the refinement process will contain the set of unlabelled examples $e_1, \ldots, e_N$. On performing an informed selection on uebuf, n examples are selected and moved out of

uebuf for labelling. Once labelled these examples are transferred into tebuf and the refinement process is initiated. Thereafter, whenever tebuf becomes empty example selection is triggered, consequently further examples are selected, labelled and moved out of uebuf into tebuf. The number of examples that get selected from uebuf need not be fixed and so can vary from iteration to iteration. In practice the sampling process can be repeated until:

- no further faults are exposed in the input KBS, hence no improvement in accuracy can be achieved; or
- a limit on the number of examples an expert is willing to label is reached; or
- uebuf is empty.

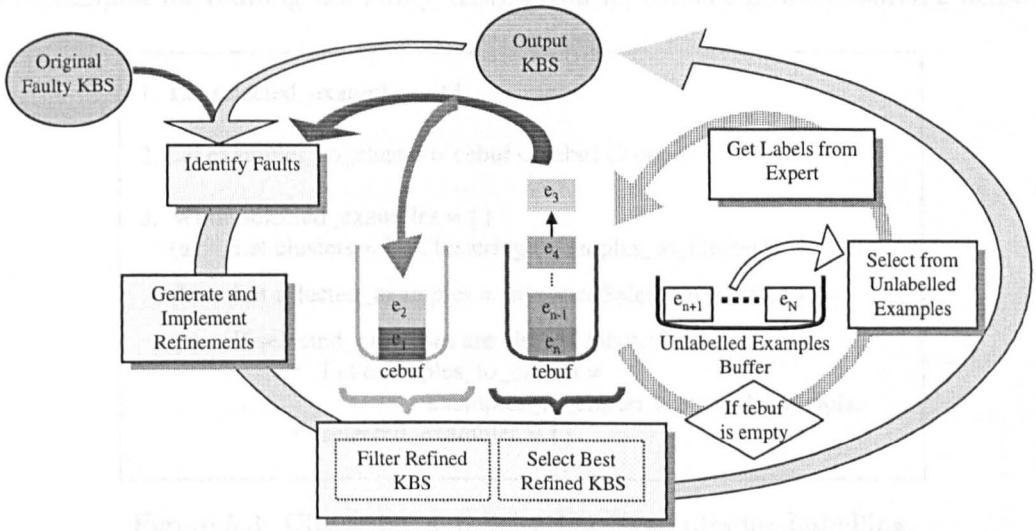


Figure 5.2: Sampling within the KRUSTtool.

So what criteria should the KRUSTtool use to decide which n examples to choose from the uebuf? Selection criteria with roots in statistical estimation techniques are increasingly being employed with encouraging experimental results for classifiers (Cohn, Ghahramani & Jordan 1996). However, the use of examples for training classifiers differs from their use for iterative refinement. In refinement:

- examples are used to expose faults in an existing KBS, and so are employed to refine incomplete concepts and not learn from scratch (Langley, Drastal, Rao & Greiner 1994); and

- examples are used for refining KBSs that model, not only classification tasks but also design tasks (Boswell et al. 1997) and even planning tasks (Tallis & Gil 1999, Gil 1995).

Direct application of currently available selection methods for learning classifiers to refinement tools, is therefore not straightforward. Moreover, unlike for learning algorithms, the input faulty KBS is an important source of information that can be exploited when selecting examples for iterative refinement. We know that the relevance of training examples for refinement, changes as refinement progresses. As the problem-solving behaviour of the faulty KBS is incrementally improved, examples that exposed faults before are less likely to expose new faults in future iterations. Meanwhile examples that did not expose faults before, may do so in future iterations. Therefore we need selection mechanisms that target examples for refining the faulty KBS, given its current problem-solving behaviour.

1. Let selected_examples = { }

2. Let examples_to_cluster = cebuf ∪ tebuf ∪ uebuf

3. While selected_examples = { }
   (a) Let clusters = DoClustering (examples_to_cluster)
   (b) Let selected_examples = InformedSelection (clusters)
   (c) If selected_examples are already labelled then
       * Let examples_to_cluster =
         examples_to_cluster \ selected_examples
       * selected_examples = { }

Figure 5.3: Clustering and Selecting Examples for Labelling.

Figure 5.3 outlines the approach that is adopted for refinement example selection. The available examples (i.e. cebuf ∪ tebuf ∪ uebuf) are partitioned into clusters. Here an unsupervised learning approach is required since we are dealing with unlabelled examples. With increased selection iterations the selected examples in step 3(b) can in some instances be themselves labelled, therefore in such circumstances further iterations will need to be triggered until one or more unlabelled examples are selected. This can be avoided by restricting `examples_to_cluster` to just unlabelled examples (in uebuf). However, doing so will increase the number of singletons, thereby reducing selection decisions that can be made on the basis of intra-cluster example relationships. For knowledge refinement

purposes the clustering and subsequent example selection from clusters must exploit the relationship between examples with respect to how they are solved by the current KBS, instead of existing sampling techniques that exploit the statistical distribution of examples.

5.2 Selection Guided by Problem-Solving Behaviour

Clustering involves the formation of distinct example clusters, by grouping similar examples according to a pre-determined similarity distance metric (Rasumssen 1992, Hanson 1990, Kodratoff 1988). There are two main clustering approaches:

- non-hierarchical clustering, where heuristics are used to group examples into one of several pre-determined clusters; and
- hierarchical clustering, where similar examples or clusters are recursively fused together, forming several nested clusters.

Non-hierarchical clustering requires initial knowledge about the number of classes, or alternatively, knowledge about the classes in the problem domain (Michalski & Stepp 1990, Fisher 1985, Michalski & Stepp 1983). Usually, with knowledge refinement there is no prior knowledge about the number of faulty areas in the KBS, far less the types of faults that need to be addressed. Therefore we employ hierarchical clustering. where a *similarity metric* needs to be defined before a *clustering technique* can progressively develop the clusters.

5.2.1 Similarity Metric

The KRUSTtool records the problem-solving that is undertaken by a KBS for an example in the positive problem graph (see Section 4.1.2). This graph records the rule activations and the order in which these activations occur. Therefore we can use the similarity between the positive problem graphs of examples, to determine which examples trigger similar problem solving behaviour in the faulty KBS. The task of establishing similarity in this manner, means that we need only be interested in rule activations for examples, regardless of whether or not the system solution is correct. More importantly, examples need not be labelled for this task.

Given a KBS containing rules $R_1, \ldots, R_m$, we define a binary valued *rule vector* corresponding to an example e, as $\mathbf{r} = (r_1, \ldots, r_m)$ where $r_i = 1$ if R_i appears in the problem graph for e, and $r_i = 0$ otherwise. Thus, in Figure 5.4, the rule vector for the positive problem graph of Example A is (0, 0, 0, 1, 0, 0, 1, 1, 1, 0), and for Example B is (0, 1, 1, 1, 1, 0, 0, 0, 0, 0), where m=10. Presently, we are only interested in the activation of a given rule regardless of the number of times it activated or when it activated.

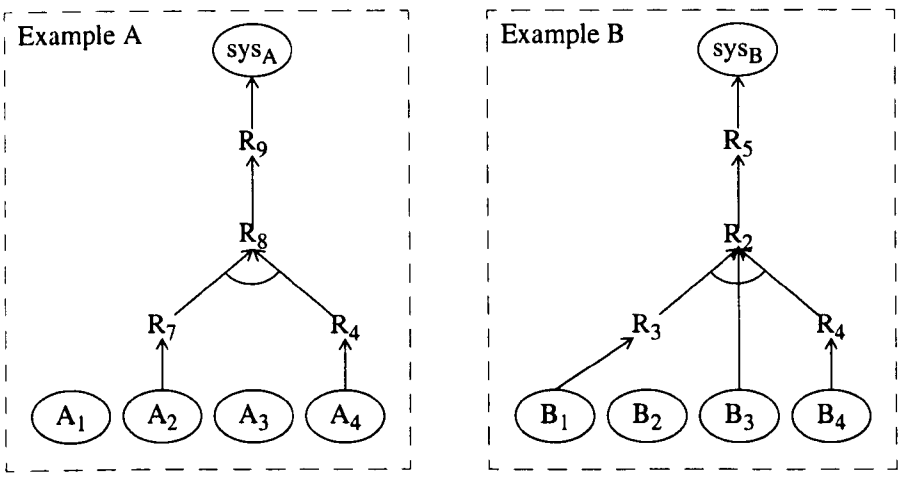


Figure 5.4: Positive Problem Graphs for Example A and Example B.

The similarity measure needs to capture refinement similarity between two unlabelled training examples e_1, e_2. As refinement similarity depends on the similarity in problem solving behaviour, the similarity between e_1, e_2, can be established by comparing their rule vectors $\mathbf{r}_1, \mathbf{r}_2$. For this purpose the Euclidean distance metric may be used, but it can lead to two rule vectors being regarded as highly similar, despite them having no common rule activations. Association coefficients (Willett 1988, van Rijsbergen 1980) avoid this by focusing on the common rule activations, and normalizing by the number of rule activations in both rule vectors, thereby ignoring rules that are not activated. We employ the Dice coefficient, a commonly used similarity measure of this type:

$$RefSim(e_1, e_2) = Dice(\mathbf{r}_1, \mathbf{r}_2) = \frac{2\ \mathbf{r}_1 .\ \mathbf{r}_2}{\mathbf{r}_1 .\ \mathbf{r}_1 + \mathbf{r}_2 .\ \mathbf{r}_2}$$

Accordingly, the similarity between examples A and B in Figure 5.4 is 0.25.

5.2.2 Clustering Technique

We use an agglomerative hierarchical clustering technique, where those training examples with the greatest similarity are united in small clusters. These clusters are then iteratively fused, until inter-cluster dissimilarity achieves a predetermined threshold. There are three commonly used approaches to cluster fusion (Hanson 1990).

Nearest neighbour method: where those two clusters that have the minimum distance between their most similar cluster members are fused. This form of cluster fusion tends to over-generalise and create fusion where there should not be any.

Farthest neighbour method: where those two clusters that have the minimum distance between their most dissimilar cluster members are fused. Typically, this form of cluster fusion leads to small, tightly bound clusters.

Centroid Method: where clusters are fused based on the average pairwise distance. This form of fusion can be seen as an intermediate of the two previous methods.

The farthest neighbour approach is chosen because it is most sensitive to dissimilarities between examples, and forms cohesive clusters that are better able to represent the different problem solving areas of the current KBS. Important to hierarchical clustering is the cluster fusion stopping threshold, referred to as the *cluster threshold.* It is this threshold that terminates the recursive fusion during clustering. A high cluster threshold leads to over generalisation, and to over specialisation when set too low. Extreme situations occur when the recursive cluster fusion process terminates once the complete set of training examples are contained within a single cluster, or when the number of clusters equals the number of training examples. Therefore, selecting a suitable threshold must be approached with caution.

Consider the clustering of 37 examples according to rule activations with the Soybean KBS in Appendix B.1 (see Figure 5.5). The small squares contain the calculated distance between two clusters while the rectangles denote the 12 clusters formed with the cluster threshold set at 0.26, i.e., fusion takes place only if inter-cluster dissimilarity is below 0.26. In the left most corner, training examples `plant751`, `plant29` and `plant21`, have identical rule activations so they have a dissimilarity score of 0. These examples are fused together with example `plant94` as the dissimilarity between the farthest neighbours is less than

0.26. The dissimilarity scores that stop further fusion is highlighted in bold font. For instance the cluster in the right most corner containing **plant491** is not fused with the cluster containing **plant745**, **plant697** and **plant700**, because the dissimilarity score of 0.29 is greater than the cluster threshold.

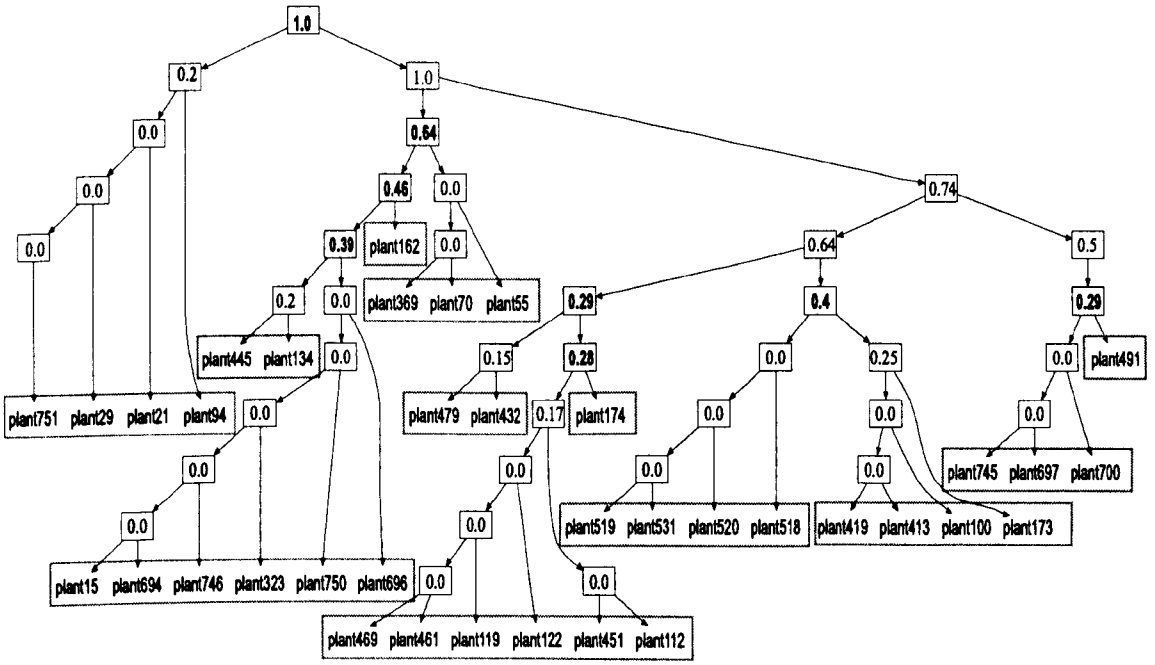


Figure 5.5: Clustering of 37 examples from the Soybean domain.

5.3 Selecting Examples using Clusters

Clusters allow a more informed selection choice than a random selection of examples. Each cluster represents the problem-solving behaviour pertaining to some part of the faulty KBS, because examples with similar rule activations are clustered together. If we happen to know which area of the KBS is faulty, the task of example selection is reduced to picking the cluster related to that area. However, in most cases the KRUSTtool has no prior knowledge about what parts of the KBS might be faulty, therefore, we need a more general selection technique that targets all potentially faulty parts of the KBS.

Since each cluster contains examples which are solved in a similar way by the KBS, it might appear reasonable to assume that refining a fault exposed by a single example from a cluster would correct the rest of the cluster. One selection method CLUSTERREP exploits this assumption by randomly selecting a single unlabelled example from each cluster, with

the aim of selecting a subset of examples representative of the problem solving behaviour of the faulty KBS. Therefore, according to CLUSTERREP, 12 examples will be selected from the clustering scenario in Figure 5.5, where a single example is randomly selected from each of the 12 clusters. Certainly, training examples that activate several rules in common appear in the same cluster and typically are also similar in their observables. However, in some situations examples from a single cluster may not have similar observables, and so may contain a pair of examples where:

- a possible refinement for one example results in another fault for which there is no obvious refinement; or
- a possible refinement for one example introduces a fault into the solution of the other.

Faults of this nature are termed *interacting faults* and the involved pair of examples is a conflict pair that triggers backtracking. In Chapter 3, we found that *conflict pairs* improve refinement accuracy and guides refinement search to the best incremental refinements. Therefore, if *conflict pairs* do get clustered together we need *informed selection heuristics* that are able to identify and select these pairs.

5.3.1 Interacting Faults

```
(defrule R1
   (filed_for_bankruptcy ?Student) (enlisted ?Student)
   => (assert (financial_deferment ?Student)))

(defrule R2
   (disabled ?Student) (filed_for_bankruptcy)
   => (assert (disable_deferment ?Student)))

(defrule R3
   (financial_deferment ?Student)
   => (assert (eligible_for_deferment ?Student)))

(defrule R4
   (disable_deferment ?Student)
   => (assert (eligible_for_deferment ?Student)))
```

Figure 5.6: Four rules taken from a corrupted student loans advisor in Clips.

We use four Clips rules taken from a corrupted version of a student loans adviser to demonstrate interacting faults and their effect on refinement generation. Of these rules, two have been corrupted by adding extra conditions, highlighted in bold (see Figure 5.6). Here, R_1 translates to "if a student has filed for bankruptcy and is enlisted in a military organisation then grant the student a financial deferment", and R_2 translates to "if a student is disabled and has filed for bankruptcy then grant the student a disability deferment". Assume that the KRUSTtool is attempting to fix these rules based on fault evidence provided by training examples e_1 and e_2 in that order.

e_1 = ⟨[(**filed_for_bankruptcy** $\mathbf{id_a}$), (**enrolled uci** 4)], (**eligible_for_deferment** $\mathbf{id_a}$)⟩

e_2 = ⟨[(**disabled** $\mathbf{id_b}$), (**enrolled uci** 5)], (**eligible_for_deferment** $\mathbf{id_b}$)⟩

Example e_1 concerns a student at uci that has filed for bankruptcy and according to the expert should be eligible for deferment. However, when reasoning with the faulty rules the system solution will not match that of the expert's, as the corruption to R_2 prevents it from activating. Therefore, the KRUSTtool will attempt to refine the faulty rules by either generalising R_1 or R_2, by deleting condition `(enlisted ?Student)`, or `(disabled ?Student)`, respectively.

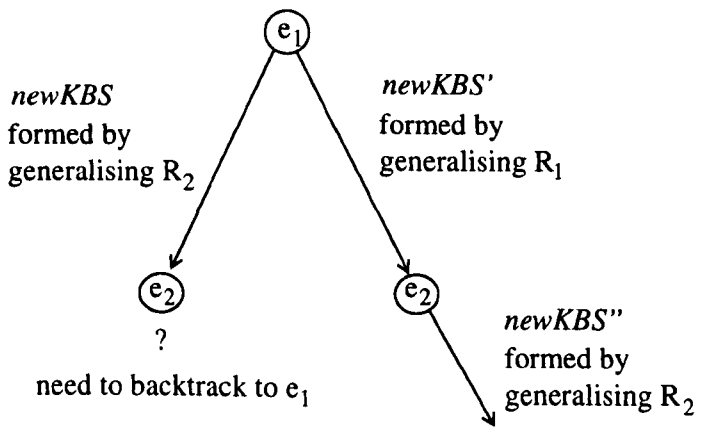


Figure 5.7: Non-optimal refinement choice triggers backtracking.

Let us assume that the KRUSTtool chooses to refine by incorrectly generalising R_2 (instead of R_1), and implements this as *newKBS* (Figure 5.7). On proceeding to the next refinement cycle with *newKBS* as the input KBS, the KRUSTtool is presented with fault evidence from training example e_2; a disabled student enrolled at uci and eligible for

deferment. A direct consequence of generalising R_2 is that the KRUSTtool is now left with no obvious refinement that can fix the fault exposed by example e_2. Consequently, it is forced to re-think its previous refinement choice of generalising R_2 instead of R_1, and so faces the prospect of re-starting refinement from a previous state. Notice that if R_2 and R_1 were corrupted, but had no common condition that matched observables from either e_1 or e_2 (for instance like `filed_for_bankruptcy`) then the faults exposed by e_1 and e_2 in Figure 5.6 would not be interacting.

```
(defrule R5
   (longest_absence ?abs_units)
   (enrolled ?en_units)
   (test (< ?abs_units 5))
   (test (>= ?en_units 5))
   => (assert (no_payment_due)))

(defrule R6
   (enrolled ?en_units)
   (test (> ?en_units 15))
   => (assert (no_payment_due)))

(defrule R7
   (not (no_payment_due)) => (assert (payment_due)))
```

Figure 5.8: Three rules taken from a corrupted student loans advisor in Clips.

We use three different rules in Figure 5.8 to demonstrate how a selected refinement has the effect of introducing a new fault that interacts with an existing fault. Here rule R_6 which translates to "if a student is enrolled and the number of units enrolled is greater than 15 then payment is not due" has been corrupted by introducing an incorrect comparison value of 15. Assume that the KRUSTtool is attempting to fix these rules based on fault evidence provided by training example e_3 and e_4 in that order.

$$e_3 = \langle[(\texttt{longest\_absence}\ \texttt{id}_c\ \texttt{ucla}\ 5), (\texttt{enrolled}\ \texttt{ucla}\ 5)], (\texttt{payment\_due})\rangle$$

$$e_4 = \langle[(\texttt{longest\_absence}\ \texttt{id}_d\ \texttt{ucla}\ 5), (\texttt{enrolled}\ \texttt{ucla}\ 12)], (\texttt{no\_payment\_due})\rangle$$

Example e_3 concerns a student at ucla enrolled in 5 courses and absent for 5 days. Since (`no_payment_due`) will not be asserted (because R_5 and R_6 cannot activate) R_7 will be activated concluding `payment_due`. This assertion matches the expert's solution

therefore refinement is not needed. In Chapter 3 examples (like e_3) that did not trigger refinement were referred to as latent examples. The next refinement example is e_4, but now the system solution will incorrectly conclude **payment_due**, because R_6 is too specialised and will not activate. Refinement will be triggered and the KRUSTtool will attempt to refine the rules by either generalising R_5 or R_6, by changing the comparison operator to $\leq$; or by changing the comparison value to 11. Assume that it incorrectly selects the

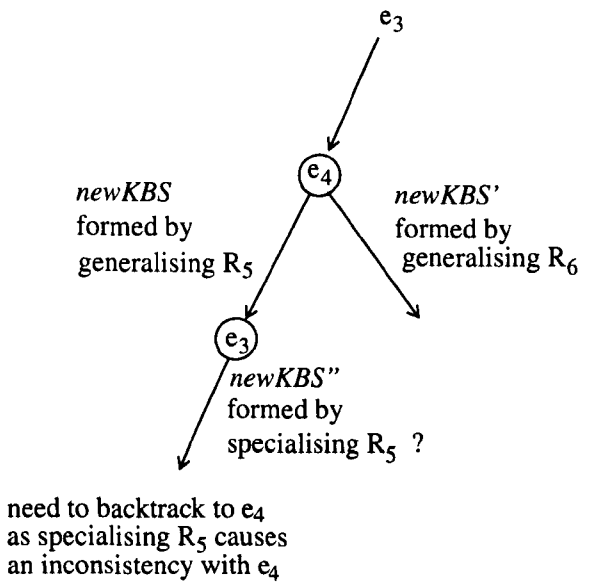


Figure 5.9: Non-optimal refinement choice activates latent example and triggers backtracking.

refinement to R_5, which would then activate the latent example e_3 (see Figure 5.9). On activation, example e_3 will drive the next refinement cycle. In that cycle the KRUSTtool will need to specialise R_5, because it is now too generalised and incorrectly concludes **no_payment_due** for example e_3. However, specialising R_5 will undo e_4's refinement of generalising R_5, consequently, backtracking is triggered resuming the refinement process from e_4 with $newKBS'$ generalising R_6. Notice, that although we had started with a single corruption to R_6, selecting a non-optimal refinement introduced a second fault that interacted with this initial corruption, resulting in backtracking.

The presence of interacting faults affects the refinement process, because selecting a refined KBS in a previous iteration can cause refinement conflicts in a subsequent iteration. These conflicts can only be detected subject to the availability of fault evidence provided by a pair of examples, a conflict pair (such as e_1 and e_2, or e_3 and e_4, above). If a cluster

contains conflict pairs like these, we would want to select further examples from this cluster. In these situations CLUSTERREP is not sufficient as it randomly selects a *single* example from each cluster, thereby ignoring all other examples in that cluster, including conflict pairs. A mechanism is needed to identify conflict pairs when they occur in the same cluster so that we ensure that examples exposing interacting faults are chosen. This necessitates an investigation of the problem-solving behaviour of labelled conflict pairs that occur in the same cluster. The aim of such an investigation is to establish criteria that would enable the identification and selection of conflict pairs from a cluster when they are still unlabelled.

5.3.2 Characteristics of Conflict Pairs

An analysis of *labelled* conflict pairs revealed that they tend to have overlapping positive problem graphs, yet the best refinement choices for the pair are distinguished from each other. Essentially their proofs may utilise similar parts of the KBS but their best refinement exercises separate parts. Figure 5.10 shows the problem-solving for a hypothetical conflict pair, C=$\langle[C_1,\ldots,C_6]\,|goal_C\rangle$ and D=$\langle[D_1,\ldots,D_6]\,|goal_D\rangle$. The darkened arrows and bold rule names highlight the positive problem graphs for examples C and D; i.e. the rules that are activated by the observables for each example. Each example has resulted in the activation of the same end rule R_3, but the solutions (sys_C and sys_D) might occur with different variable bindings. Typically conflict pairs tend to have a substantial area of the positive problem graph in common. Consequently, they tend to be placed in the same cluster, and easily mistaken as representing the same fault.

Figure 5.10 also shows all rules that might have concluded each target goal if they had been activated; i.e. the negative problem graph. With example C, R_5 is only partially satisfied by R_1's conclusion. The arrow from C_4 is fainter to indicate that the condition in R_5 is not met by this observable without the condition being generalised somehow. The other possible route via R_4 requires both of its conditions to be generalised before being satisfied by C_5 and C_6. Possible refinements attempt to specialise rules in the positive problem graph and generalise those from the negative problem graph[1]. However, specialising R_2 to disallow the proof of sys_C for example C may cause problems when

[1]For a description of KRUSTtool's specialisation and generalisation refinement operators see (Boswell & Craw 1999).

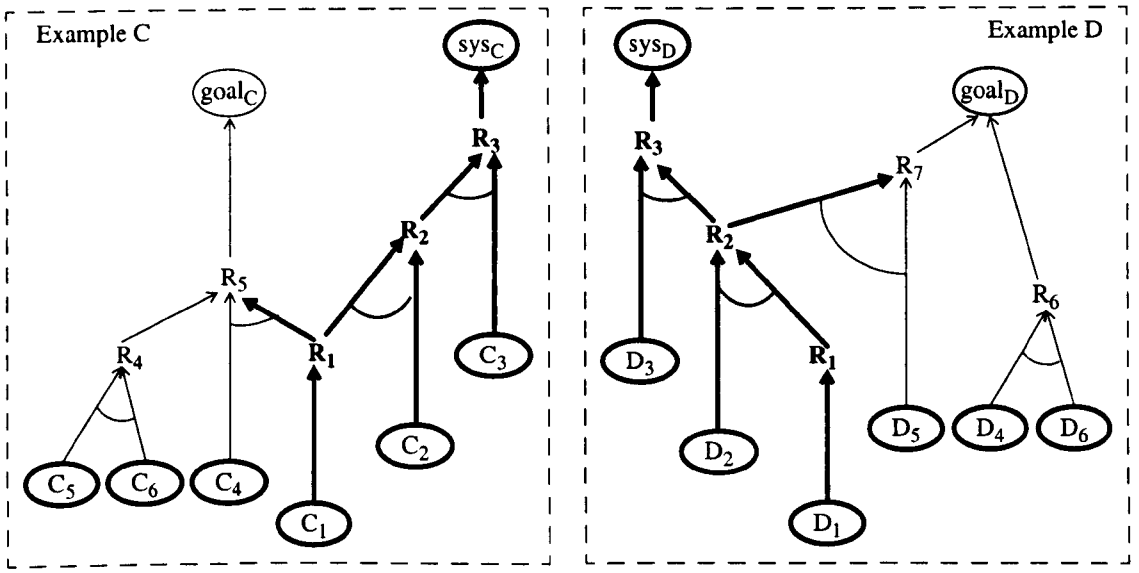


Figure 5.10: Over lapping problem graphs for conflict pairs.

generalising R_7 to allow the proof of $goal_D$, for example D, and vice versa with R_1 and R_5. Essentially, even though conflict pairs are clustered together, a refinement for one example will not necessarily repair the other; i.e. their negative problem graphs are fairly disjoint.

5.3.3 Informed Selection Heuristics

When examples are *unlabelled* we do not know the goals and cannot build the negative problem graphs as in Figure 5.10. Instead we identify potential conflict pairs by formulating an indirect estimate of how overlapping the two negative problem graphs might be. For this purpose we:

- compute the dissimilarity of examples based on their observables since the non-activations in the negative problem graph depend on these observables; and
- use this dissimilarity to compute the intra cluster dissimilarity score for each cluster which can then be exploited by informed selection heuristics.

Observable-Based Dissimilarity of Examples

The dissimilarity between two examples, $e_a = \langle [f_a^1, \ldots, f_a^m], ? \rangle$, and $e_b = \langle [f_b^1, \ldots, f_b^n], ? \rangle$, is computed by comparing each of their corresponding observables, where the number of

observables m and n are not necessarily equal.

$$dissimilarity(e_a, e_b) = \frac{\sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}\Delta^2(f_a^i, f_b^j)}}{2 * (m+n)} \quad (5.1)$$

The maximum possible dissimilarity distance between examples e_a and e_b is $m+n$, so the actual distance can be normalised by dividing by $m+n$. However, since the observable distances are counted twice, the actual distance is normalised by $2 * (m+n)$.

The difficulty with calculating Δ is that each observable can be represented as either an object-attribute-value (OAV) or as an ordered-term (OT). For instance the observable related to a student x, enrolled in 5 units can be represented as, (`enrolled`, x, 5), which is an OAV. Applying functions *value*, *object* and *attribute* to this OAV, returns 5, x and `enrolled`, respectively. Now, consider the OT representation of an observable related to a student x, in school s, and absent for 10 units: (`longest_absence`, x, s, 10). As before function *attribute* will return `longest_absence`, however, unlike OAVs, OTs can have one or more values such as s and 10, additionally, it may or may not contain an object, such as x. Here, we assume that 3 consecutive calls to function *term* when applied to (`longest_absence`, x, s, 10), returns x, s followed by 10, and function *length* returns 3. Essentially, *length* returns the number of calls that should be made to *term*, in order to access each of the values in the OT. For an OAV *length* is always 1, and a single call to *term* is sufficient to access the OAV value.

We need additional meta-knowledge specifying which terms should and should not be considered in an OT, and which OAVs should contribute towards dissimilarity. For instance with OT (`longest_absence`, x, s, 10), the object x will be a student identification number and should typically not contribute towards a dissimilarity score, while s and 10 should. To this end we maintain meta-knowledge in the form of a binary vector where a 1 indicates that the term in the corresponding position should contribute towards dissimilarity, and 0 otherwise. For instance *vector*(`longest_absence`, x, s, 10) = (0 1 1), specifies that s and 10 should contribute towards dissimilarity while x should not. Similarly for the OAV (`enrolled`, x, 5) a vector of (0 1) specifies that only 5 should contribute towards dissimilarity.

We can now use functions *attribute*, *length*, *term* and meta-knowledge *vector* to define

the dissimilarity between two observables, Δ.

$$\Delta(x,y) = \begin{cases} 0 & \text{if attribute}(x) \neq \text{attribute}(y) \\ \delta(x,y) & \text{if attribute}(x)=\text{attribute}(y) \end{cases} \tag{5.2}$$

The matching of attributes in δ is important to ensure that dissimilarity is calculated between observable pairs with matching attributes because it does not make sense to do so for non-matching observables.

$$\delta(x,y) = \sqrt{\sum_{l=1}^{length(x)} vector_l(x) * \delta_T^2(term_l(x), term_l(y))} \tag{5.3}$$

$$\delta_T(x,y) = \begin{cases} 0 & \text{if } x=y \\ \|x-y\| & \text{if } x \text{ and } y \text{ are numeric} \\ 1 & \text{otherwise} \end{cases} \tag{5.4}$$

In δ_T, $\|\ \|$ indicates that the difference is normalised by the maximum and minimum value difference.

Intra Cluster Dissimilarity

In order to calculate the intra cluster dissimilarity (ICD) for a cluster $C=\{e_1,\ldots,e_n\}$ we first calculate the *Dissimilarity* score for each example in C. *Dissimilarity* of example e_i is simply the sum of all pair-wise dissimilarities between e_i and the remaining examples in C.

$$Dissimilarity(e_i, C) = \sum_{e_j \in C} dissimilarity(e_i, e_j)$$

The ICD score for cluster C containing $|C|$ examples, is the average *Dissimilarity* of all its examples.

$$ICD(C) = \frac{\sum_{e_i \in C} \sum_{e_j \in C} dissimilarity(e_i, e_j)}{|C|}$$

There is some argument for ignoring the influence of observables that have already resulted in activations when calculating *dissimilarity* between examples, however, as the contribution towards dissimilarity from observables associated with activations, compared to those associated with (non) activations is negligible, we have opted for the simpler *dissimilarity* score using all observables.

When a cluster has a high ICD score there is reason to believe that such a cluster may contain conflict pairs, and we want to select it first for refinement. The intuition behind this is that examples clustered together based on similarity of the KBS's problem solving behaviour would normally also be similar in their observables. If observables are dissimilar then it is likely that problem solving behaviour of the KBS for that cluster is faulty and additionally contains conflict pairs which necessitate the selection of more than one example to fix the associated faults. We propose the CLUSTERDISIM family of selection heuristics that pick varying numbers of examples from the cluster with the highest ICD score as follows:

- *CLUSTER selects *all* examples;
- K-CLUSTER selects the K examples with highest *Dissimilarity*; and
- >CLUSTER selects examples with *Dissimilarity* scores above a pre-determined threshold.

5.4 Experimental Evaluation

We use the student loans KBS with 5 corruptions (see Appendix A) and the Soybean KBS with 7 corruptions (see Appendix B.1). These corruptions are controlled such that interacting faults occur only with the Student loans KBS. This means that with the Soybean KBS examples from different classes always have distinct problem graphs. The types of corruptions that can be introduced are constrained by the available refinement

operators as discussed in Chapter 1. CLUSTERREP and the CLUSTERDISIM family of selection heuristics are compared against RANDOM, where refinement examples are selected randomly. The experiments test whether selective sampling produces refined KBSs with comparable accuracy but using fewer labelled examples than RANDOM.

For each domain, a set of 100 training examples and a further 100 evaluation examples are randomly selected from the data set. The KRUSTtool is run with increasing subsets of the 100 training examples. Although all examples in the data set are labelled for experimentation purposes, these labels are ignored until examples are selected from uebuf into tebuf for the refinement task. Therefore, the labelling step in the *select-label-refine* iterative process is implicit, and the stop criterion is that the refined KBS has 100% accuracy on the training examples after the refinement step. In practice this criterion is not available, as only selected training examples will be labelled, therefore, in a real environment the criteria will be the availability of the expert, or a predetermined level of accuracy on the training examples. The impact of informed selection on:

- *efficiency* is determined by the percentage of unused (unselected) examples in uebuf; and
- *effectiveness* is determined by the accuracy of the final KBS on the evaluation set.

The graphs show results averaged over 10 runs for each training set size. Significance results are based on a 95% confidence level and apply the Kruskal Wallis non-parametric test (see Appendix D). The cluster fusion threshold and the *Dissimilarity* threshold for >CLUSTER with each test domain was ascertained a priori by experimenting with varying thresholds, on a separate subset of examples.

5.4.1 Student Loans Domain

There was no significant difference between the informed selection methods and RANDOM in the accuracy of the final output KBS on the evaluation set. This suggests that all methods have similar effectiveness. Figure 5.11 shows the graph for unused percentage of examples for each of the methods. We see that CLUSTERDISIM methods have significantly higher unused percentages compared to CLUSTERREP and RANDOM (p=0.005). 3-CLUSTER overall has fared best, and on average is three times more efficient than RANDOM

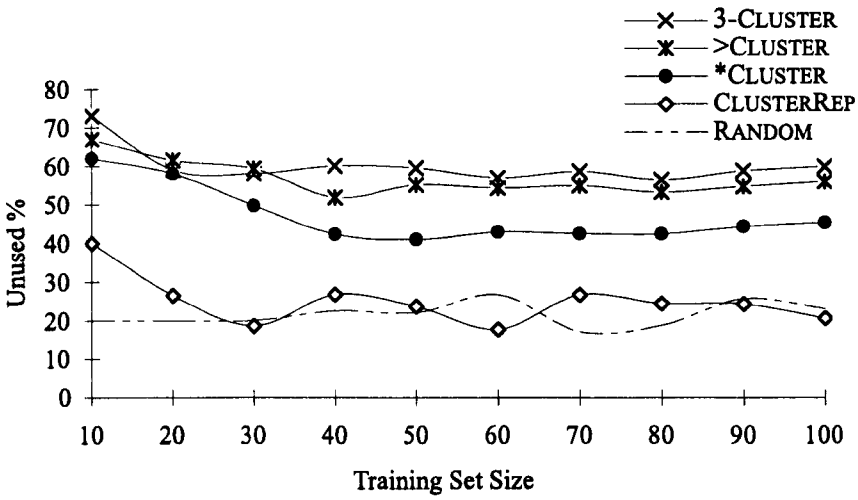


Figure 5.11: Unused examples for student loans domain.

or CLUSTERREP. 3-CLUSTER and >CLUSTER have significantly higher unused percentages compared to *CLUSTER, suggesting that the subset of most dissimilar examples from the cluster effectively targets the faults highlighted by all the examples in the cluster. All CLUSTERDISIM methods use significantly fewer training examples compared to CLUSTERREP and RANDOM. CLUSTERREP's poor performance is due to the added complication of interacting faults, and shows that selecting a single cluster representatives alone is not sufficient in these situations. The increase in unused percentage with training set size 10, seen with all methods, is explained by small training sets being insufficient to expose all faults in the KBS. As a result 100% accuracy on the training set is achieved easily, while the accuracy on the evaluation set will be significantly worse when compared to refined KBSs produced from larger training sets.

5.4.2 Soybean Disease Domain

Again there was no significant difference in accuracy and a significant difference in unused percentages (p=0.005). From the efficiency view, in this domain, CLUSTERREP, uses significantly fewer examples than *CLUSTER and RANDOM (see Figure 5.12). The success of CLUSTERREP and the failure of *CLUSTER is explained by the absence of interacting faults in this rule base. This confirms that in the absence of interacting faults we can rely on selecting a single example per cluster, as each cluster represents a distinct aspect of the faulty KBS's problem solving behaviour. A further observations is that the performance

of CLUSTERREP improves with increased training set sizes, indicating that it was able to target few, yet good, examples. Closer examination of test runs with set sizes 70, 80, 90 and 100, revealed that the number of clusters tends to be constant while the size of clusters increases with the increasing number of examples, therefore CLUSTERREP selects the same number of examples regardless of the size of the training set. On average CLUSTERREP is three-times more efficient than RANDOM or *CLUSTER. *CLUSTER's bad performance with larger training set sizes clearly shows that the absence of an appropriate selection mechanism can result in ultimately using all the unlabelled examples. The results for 3-CLUSTER and >CLUSTER methods which are derivatives of *CLUSTER, have not been plotted as they performed poorly.

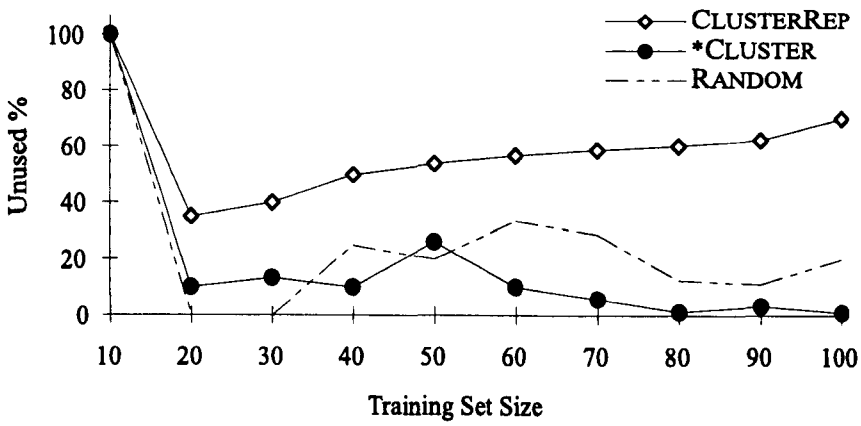


Figure 5.12: Unused examples for soybean disease domain.

5.5 Conclusion

Experimental results show that selective sampling can significantly reduce the number of examples utilised, without any penalty on the final accuracy. The refinement process was able to target particular faults that improved the accuracy of the refined KBS in a way that was effective in general. Not only did this reduce the number of refinement cycles required to achieve a particular level of competence, but it also reduced the demand on the expert's time. The selection was done based on features of the problem-solving behaviour alone and so the expert was consulted about only the selected examples. Once labelled, the selected examples were presented to the KRUSTtool.

The rule vector representation of the positive problem graph provided a simple sim-

ilarity measure that created clusters of examples that had been solved by the KBS in a similar way. This clustering was helpful in determining examples that might indicate the same refinement. A potential problem with representing examples as rule vectors is the high computational cost that might be incurred with large and complex KBSs, because each example must be executed on the KBS before the rule vector can be formed. The faulty KBSs employed in this chapter are relatively small, therefore for complex KBSs it will be important that the rule vector representation be translated into some form of feature vector representation. For instance each rule can be viewed as a correlation between a subset of observables. The goal is to identify these observable subsets where a single observable can be in none, one or more subsets. Thereafter, clustering can be based on similarity between several subsets of correlated observables. Essentially similarity in a single observable subset, corresponds to similarity in a single rule activation.

Selective sampling is important for knowledge refinement whether or not labelled training examples are plentiful. If labels are hard to obtain then it is certainly useful to identify relevant problem-solving tasks that should be labelled by the expert and then used as training examples for refinement. Conversely if there are many labelled training examples then, given that the refinement process is quite computationally expensive, it is convenient to target those examples whose refinements also fix other wrongly solved examples without further refinement, thereby reducing the number of refinement cycles. Selective sampling addresses both these issues by identifying the examples most likely to solve others that indicate the same general fault. Given the encouraging results with respect to active selection of refinement examples in this chapter, we look at active selection of examples for the filtering task in the next Chapter.

Chapter 6

Informed Selection of Filter Examples

In each refinement cycle, the refined KBS with best quality is selected from the set of proposed refined KBSs. Quality of a refined KBS is measured by ascertaining its accuracy on a set of examples, referred to as *filter examples*. This quality testing process is heavily dependent on the availability and selection of examples suited for the filtering role. In this chapter we investigate techniques that aim to actively select few yet good filter examples from the set of labelled and unlabelled examples for the KRUSTtool's filtering task.

KRUSTtool's existing filtering process and drawbacks are discussed in Section 6.1. A cluster-based approach to example selection exploiting changes in problem solving behaviour is introduced in Section 6.2, followed by an ensemble-based approach in Section 6.3. Experimental results on two domains are analysed in Section 6.4, and Chapter conclusions appear in Section 6.5.

6.1 Filtering Refined KBSs

The KRUSTtool's refinement algorithm employs several KBS filters to select the *best* refined KBS from the set of proposed refined KBSs (Craw 1996, Palmer & Craw 1996). These filters form several levels and at each level zero or more refined KBSs will successfully pass through. Essentially the successful candidates from one filter level become the input at the next subsequent filter level. In this manner the filters attempt to weed out *bad* refined

KBSs from the set of proposed refined KBSs. The eventual aim is to identify the *best* refined KBS.

Figure 6.1 illustrates three filter levels. The initial *Consistency Filter* ensures that consistency is maintained with previously solved refinement examples in cebuf, while the *Accuracy Filter* makes judgments about the quality of proposed refinements based on accuracy on a subset of training examples. Refined KBSs with highest accuracy on this subset pass on to the next level. Any ties are broken randomly. Notice, backtracking will be triggered when all refined KBSs fail to pass the consistency filter.

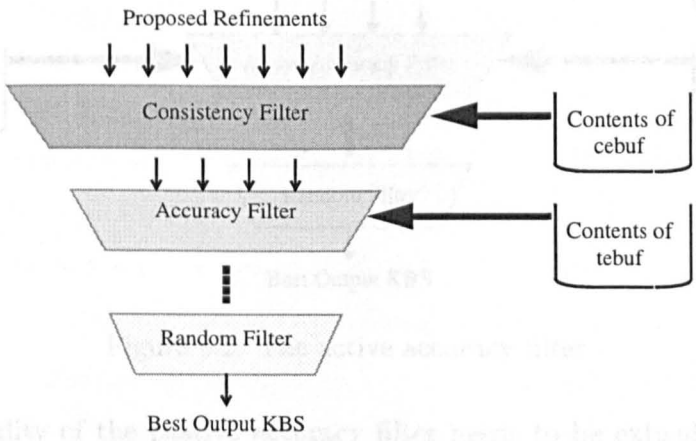


Figure 6.1: The KRUSTtool filter hierarchy.

Ensuring consistency with previous refinement examples is straightforward, as we already know which training examples have previously triggered refinement and have been solved correctly (these are constraint examples in *cebuf*). The difficulty with the *Accuracy Filter* is in identifying a relevant subset of examples upon which the judgment can be based. Presently, the subset is simply all labelled training examples yet to be processed in tebuf. Obvious disadvantages in such a scheme include:

- high processing costs when tebuf is large;
- insufficient evidence for judgment when tebuf is small; and
- duplication bias, where a large number of similar examples may incorrectly suggest high (or low) accuracy.

Even if the number of training examples in tebuf is not too extreme, using all training examples is not sensible as proposed refinements may have affected only a subset of these

examples. Needless to say, using un-affected examples for judgment purposes will not contribute additional information towards ascertaining whether a proposed refinement is *good* or not, but instead will increase processing costs. Moreover, confining the role of filtering to just labelled examples may mean that other relevant examples in uebuf will not be able to influence filtering.

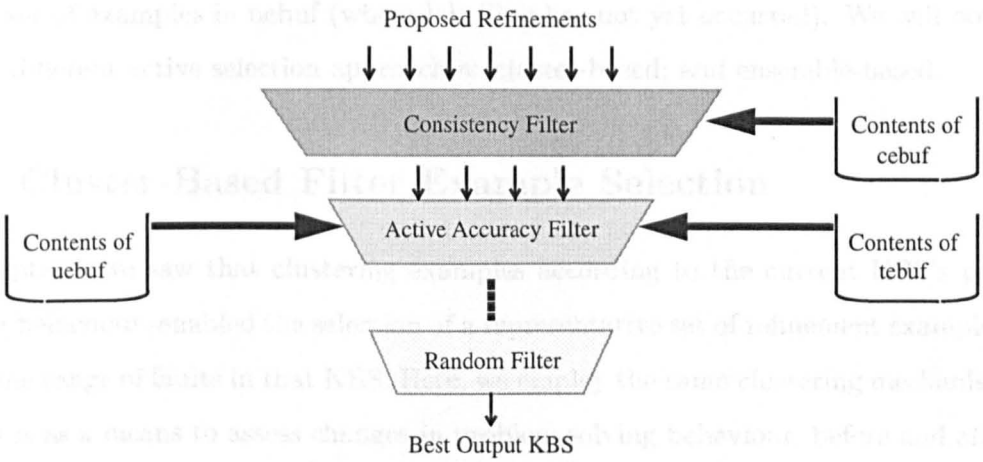


Figure 6.2: The active accuracy filter.

The functionality of the passive accuracy filter needs to be extended to one that is able to actively select relevant examples that are testing of the proposed refined KBSs. An accuracy filter with active selection capability will be referred to as the *active accuracy filter*. Figure 6.2 illustrates such a filter utilising examples from both tebuf and uebuf. It is hoped that incorporating active selection of filter examples in this manner will facilitate:

- the selection of few yet *good* examples, reducing needless processing and minimising labelling cost;
- efficiency gains by improved guidance through the space of possible refinements, thereby avoiding refinement dead-ends and reducing the need for backtracking; and
- accuracy gains by moving refinement search to parts of the search space containing more promising refined KBSs.

To achieve these goals, the active accuracy filter needs to select examples that are affected by the proposed refinement. *Affected examples* are those examples that as a result of refinement get solved differently; for instance an example previously correctly solved is now incorrectly solved or vice versa. However, things are more complicated than that, as

some of the effects are to be expected while others are not. This means that the active accuracy filter must not only identify affected examples, but select only those examples that should not have been affected the way they have. Additionally, example selection by the active accuracy filter must not be based on techniques that simply compare the system and expert solutions, because active selection of filter examples must also extend to the set of examples in uebuf (where labelling has not yet occurred). We will now look at two different active selection approaches: cluster-based; and ensemble-based.

6.2 Cluster-Based Filter Example Selection

In Chapter 5 we saw that clustering examples according to the current KBS's problem solving behaviour, enabled the selection of a representative set of refinement examples that cover the range of faults in that KBS. Here, we employ the same clustering mechanism and extend it as a means to assess changes in problem solving behaviour, before and after the proposed refinement. Changes in problem solving behaviour are captured by analysing changes in cluster membership. Essentially, examples that get clustered differently are more likely to have been affected by the refinement.

1. Cluster examples based on problem solving behaviour of the input KBS.

2. For each refined KBS that passed the consistency filter:
   (a) Repeat step 1, but this time based on problem solving behaviour of the refined KBS.
   (b) Compare example clusters formed with the input KBS to those formed with the refined KBS in step 2(a), analysing changes to cluster membership.
   (c) Identify those examples with changed cluster membership, noting them as affected examples.

3. Select filter examples from those noted as affected.

Figure 6.3: Algorithm for the Cluster-Based Approach.

The algorithm in Figure 6.3, outlines the steps involved in the cluster-based approach. In step 1, examples in tebuf and uebuf are clustered with respect to the input KBS's problem solving behaviour. The example clusters thus formed are compared with example clusters formed according to problem solving behaviour of each refined KBS in step 2. The

goal of this comparison is to identify affected examples by analysing changes in cluster membership. Instead of selecting examples simply on the basis of changes in rule activations, changes in cluster membership provides a more general view of groups of examples that are affected in similar ways. In step 3, filter examples are selected based on selection heuristics, that select from example subsets that are noted as affected.

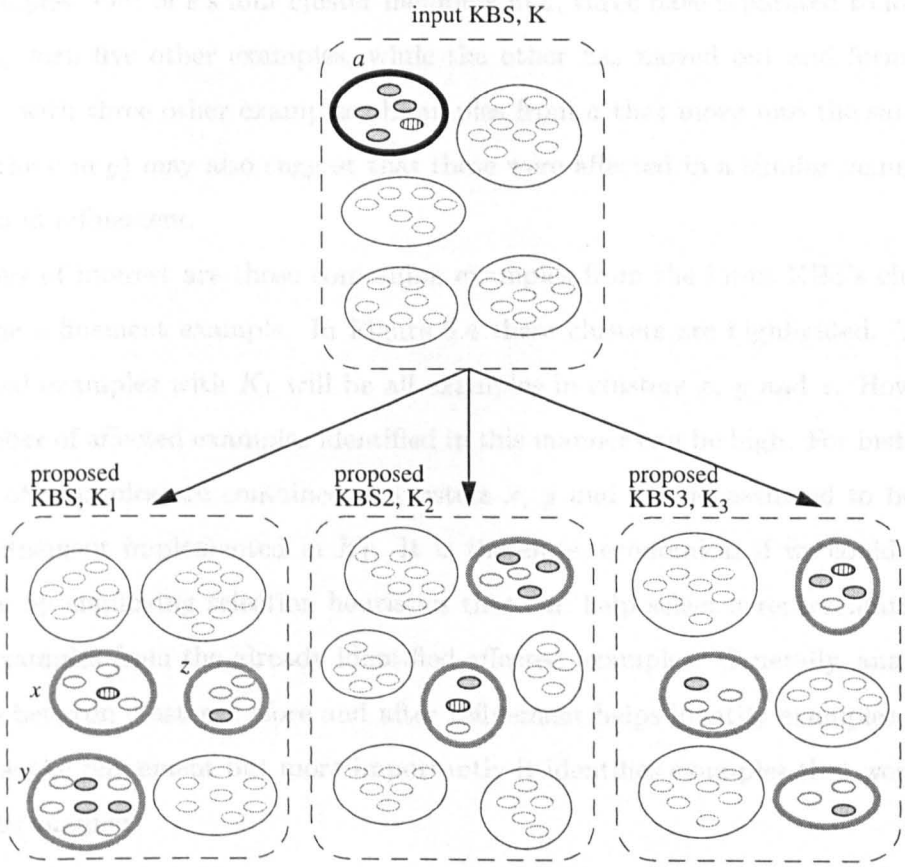


Figure 6.4: Analysing Changes in Cluster Content.

Figure 6.4 illustrates a fictitious scenario where 35 examples are clustered based on problem solving behaviour of an input KBS, K. The clustering has resulted in five example clusters, of which cluster a (bold), contains refinement example e (striped) together with four other cluster members. We refer to the 5 example clusters formed based on problem solving behaviour of K, as K's clusters. Let us assume that three refined KBSs, K_1, K_2 and K_3, generated in response to fault evidence provided by e, have already passed the consistency filter. Affected examples are identified by comparing cluster content of K's clusters with each of the refined KBS's clusters. However, a refinement can cause

significant changes in cluster content thus making the comparison difficult. Therefore, a more tractable method localises the comparison to changes relative to the original cluster in which the refinement example was a member in K's clusters, here cluster a.

In Figure 6.4 we see that according to K_1's clusters, e is no longer clustered together with it's former cluster members as in a, instead it forms a new cluster, x, with four other examples. Out of e's four cluster members in a, three have separated to form a new cluster, y, with five other examples, while the other has moved out and formed a new cluster, z, with three other examples. Examples from a that move into the same cluster (like the three in y) may also suggest that these were affected in a similar manner by the implemented refinement.

Clusters of interest are those containing examples from the input KBS's cluster containing the refinement example. In Figure 6.4 these clusters are highlighted. Therefore, the affected examples with K_1 will be all examples in clusters x, y and z. However, the total number of affected examples identified in this manner can be high. For instance with K_1, 45% of examples are contained in clusters x, y and z, and assumed to be affected by the refinement implemented in K_1. It is therefore, economical if we could cut-down this figure by employing selection heuristics that can help select a representative subset of filter examples from the already identified affected examples. Generally, analysing the difference between clusters before and after refinement helps identify examples that were affected by the refinement but more importantly it identifies examples that were affected in a similar manner.

6.2.1 Simple Selection Heuristics

Given a set of M proposed KBSs $\{K_1, \ldots, K_M\}$, we can identify M affected example sets $\{\epsilon_1, \ldots, \epsilon_M\}$. Heuristic kFILTER randomly selects k examples from each ϵ_i, resulting in $M * k$ filter examples. Any resulting duplicates are removed. A further possibility is to select the $M * k$ most frequently seen examples in $\{\epsilon_1, \ldots, \epsilon_M\}$, and we refer to this filter example selection heuristic as, FQFILTER. The advantage of both these heuristics is simplicity.

6.2.2 Refinement Extremeness Based Selection Heuristic

Selecting the best refined KBS also means that filter examples must be able to filter out refined KBSs that are too extreme, i.e. over-generalised or over-specialised. For this purpose a more targeted example selection approach is necessary, where examples although affected must only be selected as filter examples if normally they should *not* have been affected.

A KBS when generalised, typically results in new fact assertions because generalisation tries to enable rule activations which prior to refinement would not have activated. Often this amounts to weakening leaf rule conditions so that they are satisfied by the observables. Specialisation has the opposite effect to generalisation, where previously derived facts are absent after refinement. Here, instead of weakening a leaf rule's conditions they are strengthened so that observables will not satisfy one or more of the rule's conditions. With both refinement operations given an input KBS, K, and a set of proposed KBSs K_i, we wish to identify for each proposed refined KBS, K_i:

- observables that are being used differently maintaining their attributes in a list *affected attributes*, κ_i; and
- select from K_i's affected example set, ϵ_i, examples that are atypical (dissimilar) with respect to observables having matching attributes in κ_i.

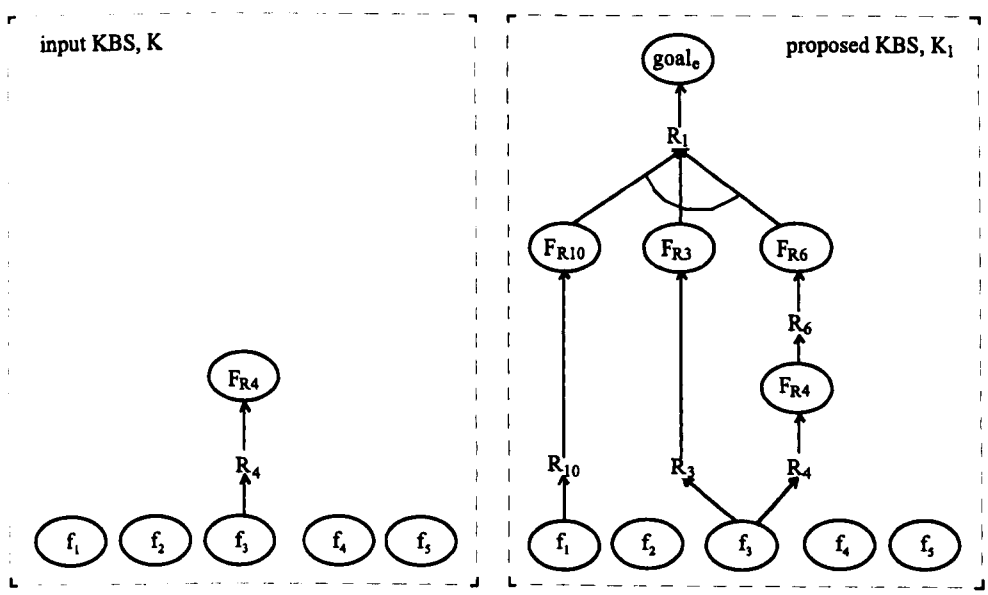


Figure 6.5: Observable usage before and after generalization.

The two positive problem graphs in Figure 6.5 illustrate the problem solving behaviour of input KBS K and a corresponding refined KBS K_1, when separately executed on refinement example e=$\langle[f_1,\ldots,f_5],goal_e\rangle$. The ovals represent observables, derived facts and the final system solution. Assume that e is a member of K_1's affected example set, ϵ_1, where K_1 is a refinement generalisation that fixes a fault in input KBS, K. Differences between K's reasoning and K_1's reasoning can be captured by examining the corresponding positive problem graphs. The differences provide information about how rule activations triggered by observables $[f_1,\ldots,f_5]$, differ between the two alternative KBS's reasoning processes. It helps us identify which observables contribute to new rule activations as a result of generalisation.

With K, we see that the single rule activation R_4 has activated because its conditions are satisfied by observable f_3, concluding F_{R4}. For the generalised K_1, R_{10} activates with f_1, R_3 with f_3 and R_6 more indirectly with f_3. Therefore, K_1's affected attribute set κ_1, will be $\{attribute(f_1), attribute(f_3)\}$. Here function *attribute* is the same as defined in Section 5.3.3. Once all examples in ϵ_1 for proposed K_1 have been analysed in this manner, κ_1 is complete.

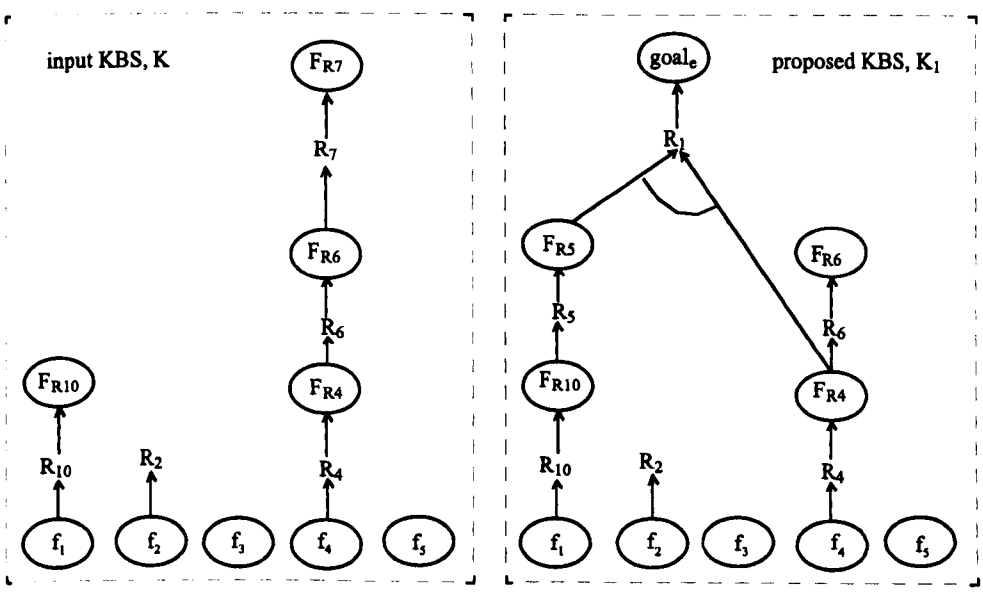


Figure 6.6: Observable usage before and after specialisation and generalisation.

Consider a different refinement scenario with both specialisation and generalisation illustrated in figure 6.6. Here, the proposed refined KBS K_1, fixes a fault exposed in input KBS K, by refinement example, e=$\langle[f_1,\ldots,f_5],goal_e\rangle$. With K, the activation of the end

rule R_7 has incorrectly concluded with system solution, F_{R7}. This is fixed in K_1 by:

- disabling rules that lead to the conclusion of F_{R7}; and
- enabling rules that should instead lead to the target goal, $goal_e$.

Consequently, we can identify several changes in observable usage with K_1 not seen with K: f_1 indirectly contributes to the activation of R_5 and so R_1; and f_4 indirectly contributes to the activation of R_1; and f_4 as a result of specialisation does not contribute to the activation of R_7. Accordingly, K_1's affected attribute list $\kappa_1 = \{attribute(f_1), attribute(f_4)\}$.

Notice that in Figure 6.5, direct analysis at the observable level was sufficient to identify changes in observable usage. However with the scenario in Figure 6.6, the effects of refinement on the reasoning process are concentrated further up the problem graph, and at first may seem not to imply any changes at lower levels. In such circumstances an analysis of changes at higher levels becomes important. Currently, the search for changes starts at the observable level and if changes are found the search stops there, otherwise, the next level of derived facts are analysed and so on.

Selecting Atypical Examples

An *Atypical* score for example e_i=$\langle[f_1, \ldots, f_m], ?\rangle$, in ϵ=$\{e_1, \ldots, e_n\}$, related to a proposed refined KBS with κ, is calculated by summing all pair-wise dissimilarities between example e_i and the remaining examples in the proposed refined KBS's affected example set, ϵ.

$$Atypical(e_i, \epsilon) = \sum_{e_i \neq e_j} dissimilarity(e_i, e_j) \tag{6.1}$$

Here, dissimilarity between examples are calculated according to equation 5.1. However, we modify equation 5.3 as follows.

$$\delta(x, y) = w(attribute(x)) * \sqrt{\sum_{l=1}^{length(x)} vector_l(x) * \delta_T^2(term_l(x), term_l(y))} \tag{6.2}$$

$$w(x) = \begin{cases} 0 & \text{if } x \notin \kappa \\ 1 & \text{if } x \in \kappa \end{cases} \tag{6.3}$$

Here, function w ensures that dissimilarity between examples is calculated only according to observables that are identified to be affected. Therefore, w returns 0 if a given observable's attribute is not in set κ, and returns 1 otherwise.

With refinements that are too extreme, it is most likely that examples with high atypical scores will be incorrectly solved by the refined KBS. Such examples have extreme values for observables that get used differently by the refined KBS as a result of the implemented refinement. Selection heuristic *FILTER, selects from each proposed refined KBS's affected example set (ϵ), k examples with highest *Atypical* scores as filter examples.

6.3 Ensemble-Based Selection

Although cluster-based filter example selection is able to identify affected examples, it is likely to be computationally very demanding. This is particularly true with increased numbers of examples and refined KBSs. The ensemble-based approach does not need to cluster examples. The refined KBSs that pass the consistency filter are used to form the ensemble, where system solutions of ensemble members are combined into a vote for or against selecting an example for filtering. Typically, we want to select examples where a majority of members are in disagreement. The underlying intuition behind this is that refined KBSs are unable to solve an example consistently when the example is particularly hard to solve and is testing of the refined KBSs. The credibility of such an approach depends on the goodness of the ensemble. Dietterich (2000), suggests that a good ensemble is one where members have an error rate of better than random guessing, and disagreement between members are uncorrelated. For filtering purposes, although the ensemble is formed by refined KBSs originating from a single input KBS, differences in system solutions is due to differences between implemented refinements alone.

Figure 6.7(a), illustrates a single refinement iteration. The corresponding ensemble formed using the refined KBSs from that iteration is in Figure 6.7(b). Here, the ensemble consists of M refined KBSs. Each member of the ensemble provides a system solution for each example in tebuf $\{e_3, \ldots, e_n\}$, and all unlabelled examples $\{e_n + 1, \ldots, e_N\}$. Since example selection is based on the degree of disagreement between ensemble members, we consider two alternative approaches to ascertaining disagreement: a heuristic approach; and a disagreement score suggested by Argamon-Engelson & Dagan (1999).

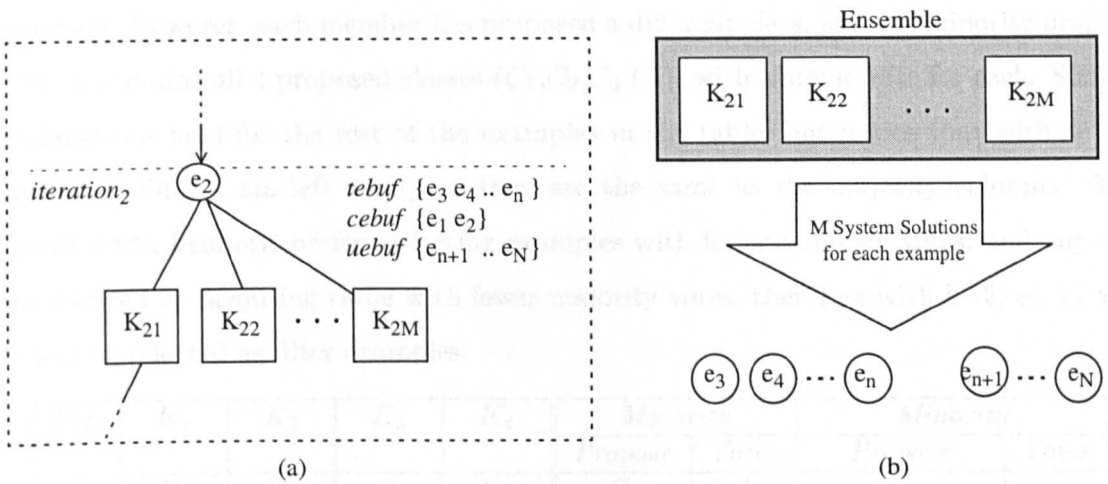


Figure 6.7: Proposed KBSs Forming an Ensemble. (a) A Refinement Iteration. (b) Corresponding Ensemble Filter.

Disagreement Heuristic

The system solutions for each example are compared noting the:

majority vote : the most frequent system solution together with the number of refined KBSs that are in agreement; and

minority vote : the least frequent system solution together with the number of refined KBSs in agreement.

With selection heuristic VOTEFILTER, the k examples with lowest minority vote are selected and any ties are resolved by favouring examples with lower majority votes.

Consider the scenario for a classification task in Table 6.1, where the KRUSTtool in response to a refinement example's fault evidence, generates several refined KBSs, of which four refined KBSs, $\{K_1, K_2, K_3, K_4\}$ have successfully passed the consistency filter. These refined KBSs form the ensemble that actively selects from examples $\{e_1, e_2, e_3, e_4, e_5\}$ based on minority and majority agreement voting. The system solution by an ensemble member (column), for an example (row), is entered in the relevant row column intersection. For instance K_1's system solution, C_1, for e_1, indicates that K_1 classifies e_1 in class C_1, while K_1, classifies e_3 in class C_1 and in class C_2. The majority of the ensemble members classify e in class C_1, hence proposing C_1 with 4 majority votes. As there are no disagreements between the ensemble members the minority columns are empty. With example e_2, the majority column is empty as there is no agreement between the ensemble

members. However, each member has proposed a different class, and the minority propose column contains all 4 proposed classes (C_1,C_2,C_3,C_4), with a single vote for each. Similar explanations hold for the rest of the examples in the table, but notice that with e_5 the minority columns are left empty as they are the same as the majority columns. The VOTEFILTER heuristic prefers selecting examples with fewer minority votes, and any ties are resolved by favouring those with fewer majority votes, therefore with k=3, e_2, e_4 and e_3 will be selected as filter examples.

Exs.	K_1	K_2	K_3	K_4	*Majority*		*Minority*	
					Propose	*Votes*	*Propose*	*Votes*
e_1	C_1	C_1	C_1	C_1	C_1	4	-	-
e_2	C_1	C_2	C_3	C_4	-	0	C_1, C_2, C_3, C_4	1
e_3	C_1,C_2	C_1,C_4	C_1,C_4	C_1,C_2	C_1	4	C_2, C_4	2
e_4	C_1,C_2	C_3	C_2,C_3	C_2,C_4	C_2	3	C_1, C_4	1
e_5	C_1	C_2	C_1	C_2	C_1, C_2	2	-	-

Table 6.1: Majority Vote by an Ensemble formed with Proposed Refined KBSs.

Establishing a majority or minority vote is difficult when one or more members of the ensemble fail to classify an example into any class. This could easily happen when proposed refined KBSs are too specialised. In such situations we could choose to ignore votes by refined KBSs that fail to classify examples. However, this may influence the selection of examples that are not necessarily ideal for filtering purposes. Instead, we allow the votes of these members on the basis of derived facts (in the absence of end facts).

Disagreement Score

A disagreement score $D(e)$, for example e, using an ensemble with M members, that classifies the example into one or more classes in C, is calculated by the entropy of the distribution of classes voted for by the ensemble members (Argamon-Engelson & Dagan 1999) (discussed in Section 2.3.2). Given the number of ensemble members classifying e in class c, where $c \in C$, denoted by $votes(c, e)$, the normalised vote entropy is:

$$D(e) = -\frac{1}{log\ min(M, |C|)} \sum_{c \in C} \frac{votes(c, e)}{M} log \frac{votes(c, e)}{M}$$

Again when refined KBSs fail to classify an example into a class, derived facts are considered instead. Therefore, the cardinality of C can change from example to example, depending on how specialised the ensemble members are, and depending on the derived facts that they conclude. Notice that the number of members is not fixed and will change from one refinement iteration to another.

The vote entropy has value 1 when all ensemble members are in disagreement, and value 0 when all are in agreement, taking on intermediate values when in partial agreement. With selection technique NTROPYFILTER, k examples with highest normalised entropy vote are selected.

6.4 Experiments

Evaluation is based on results from the Student loans test domain with a faulty KBS containing 5 corruptions (Appendix A), and the Soybean test domain with a faulty KBS containing 13 corruptions (Appendix B.2). These corruptions have been introduced according to the available refinement operators as discussed in Chapter 1. The Soybean KBS with 13 corruptions was preferred over that of just 7 corruptions (Appendix B.1), because increased corruptions are more likely to trigger backtracking. With the student loans domain we use the same experimental design of 100 evaluation and 100 training examples, and the KRUSTtool applied to increased subsets of the 100 training examples. However, with the Soybean domain the high computational costs due to the cluster-based method makes it impractical to have many repeated test runs with increased subsets of the 100 training examples. Instead, with this domain results are based on 20 test runs with 100 training and 100 evaluation examples.

The informed filter example selection heuristics KFILTER, FQFILTER, *FILTER and the ensemble-based techniques are compared against:

- NOFILTER where filter examples are examples yet to be processed in tebuf, and examples in uebuf are never selected for filtering; and
- RNDFILTER where k filter examples are randomly selected from {tebuf $\cup$ uebuf}.

The experiments investigate whether the active accuracy filter employing informed selection heuristics is able to reduce backtracking by effective refined KBS filtering that guides

the KRUSTtool through the space of possible refinements. The number of times backtracking is triggered is a good estimate of the number of refinement dead-ends encountered. Therefore, the fewer dead-ends encountered, the better the filtering heuristic at guiding refinement search. Additionally, fewer dead-ends mean fewer re-visits to previous refinement states, hence reduced iterations.

The experiments also evaluate the effect of the active accuracy filter on the error-rate of the final output KBS. It is hoped that improved guidance will move the search to parts of the refinement space resulting in higher accuracy. To ensure that any improvements are not influenced by active selection of refinement examples, the contents of tebuf are selected manually at the start of each test run. Essentially, all examples that the faulty KBS fails to solve correctly at the start of the refinement process are moved into tebuf. Manual selection of tebuf right at the start ensures that all filter example selection heuristics will have equal refinement opportunity. Additionally, this ensures that experimental results reflect the effect of filter example selection on the refinement process, decoupled from benefits from refinement example selection (in Chapter 5).

Of further interest to knowledge refinement is how useful actively selected filter examples might be for driving refinement. For this purpose a subtle difference in example buffer handling is introduced. With the Student loans domain, any selected filter examples not in tebuf once used for filtering, are moved into tebuf or cebuf accordingly. This means that filter examples will also have the opportunity to drive refinement. With the Soybean domain, actively selected filter examples are only used for filtering purposes.

6.4.1 Student Loans Domain

Figure 6.8 shows the error rate of the final refined KBS for the 5 active accuracy filter approaches. Clearly, active selection of filter examples is important; even random selection is able to significantly reduce error-rate compared to the passive NOFILTER (p=0.001). So, can a more informed selection improve on RNDFILTER's performance? Heuristics *FILTER, KFILTER and FQFILTER have significantly lower error-rates than RNDFILTER (p=0.006). The results from the ensemble-based techniques have not been plotted as they did not improve on RNDFILTER. The reason for poor performance in this domain is that disagreement amongst ensemble members is high for most examples, therefore, ties are broken randomly, reducing the performance of ensemble-based techniques to random.

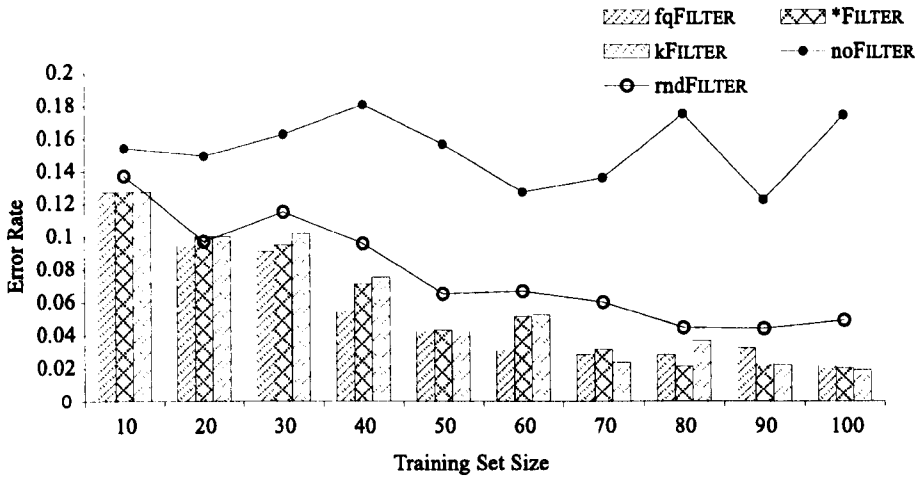


Figure 6.8: The Effects of Filter Example Selection on Error Rate.

The *FILTER heuristic undertakes the most targeted selection procedure, therefore it was surprising that there was no significant difference in error-rate between it, the KFILTER and the FQFILTER. Close examination of test runs showed that the initial manual selection of refinement examples was proving beneficial for refinement, resulting in an insignificant difference in error-rates. Therefore, a further set of experiments consisting of 20 test runs was carried out. This time the number of manually selected refinement examples at the start was halved. Of the 20 runs, the first ten involved a training and evaluation set size of 50 and the second ten a set size of 100. The results from these 20 runs indicate that *FILTER had significantly lower error-rates (p=0.03) compared to both KFILTER and FQFILTER. However, there was no significant difference between KFILTER and FQFILTER. Essentially, this suggests that atypical examples selected by *FILTER are not only well suited for filtering, but are also suited for driving refinement.

Figure 6.9 plots the number of times backtracking was triggered on encountering refinement dead-ends. Due to the ungainly performance of the ensemble-based approach on error-rate, VOTEFILTER and NTROPYFILTER are not included here. Number of backtracks triggered is significantly less with the informed selection heuristics compared to NOFILTER and RNDFILTER ($p = 0.001$). The purposeful selection of filter examples based on changes in cluster content has managed to guide the KRUSTtool through the refinement search space, reducing the need to revisit previously solved training examples. However, there was no significant difference between KFILTER, FQFILTER and *FILTER.

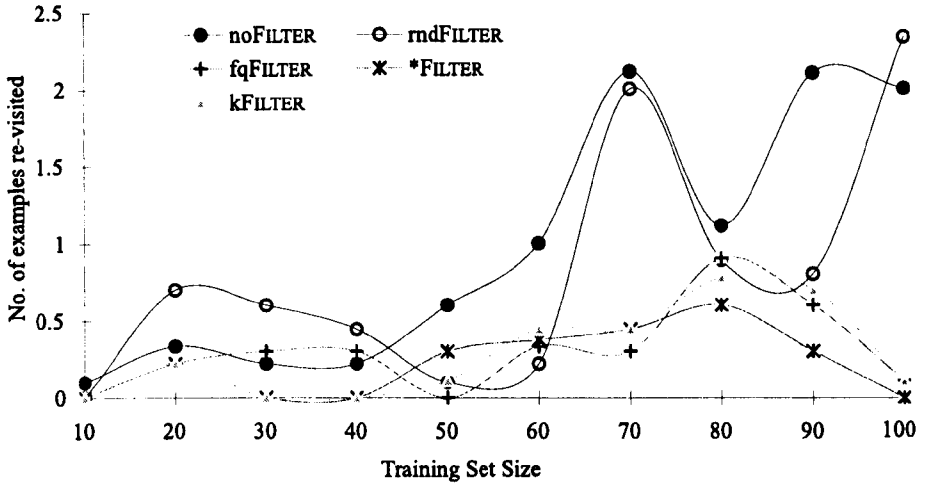


Figure 6.9: The Effects of Filtering on Backtracking.

6.4.2 Soybean Disease Domain

With experiments carried out in this domain, any actively selected filter examples once utilised for filtering, are never moved into tebuf or cebuf, therefore, filter examples do not get the opportunity to trigger refinement. The informed selection heuristics VOTEFILTER, NTROPYFILTER, *FILTER and KFILTER, were compared with RNDFILTER and NOFILTER. FQFILTER is not used here because it did not perform any better or worse than KFILTER with the student loans domain. All selection heuristics had significantly lower error-rates when compared to NOFILTER (p=0.007). However, the difference between VOTEFILTER, NTROPYFILTER, *FILTER, KFILTER and RNDFILTER is not significant.

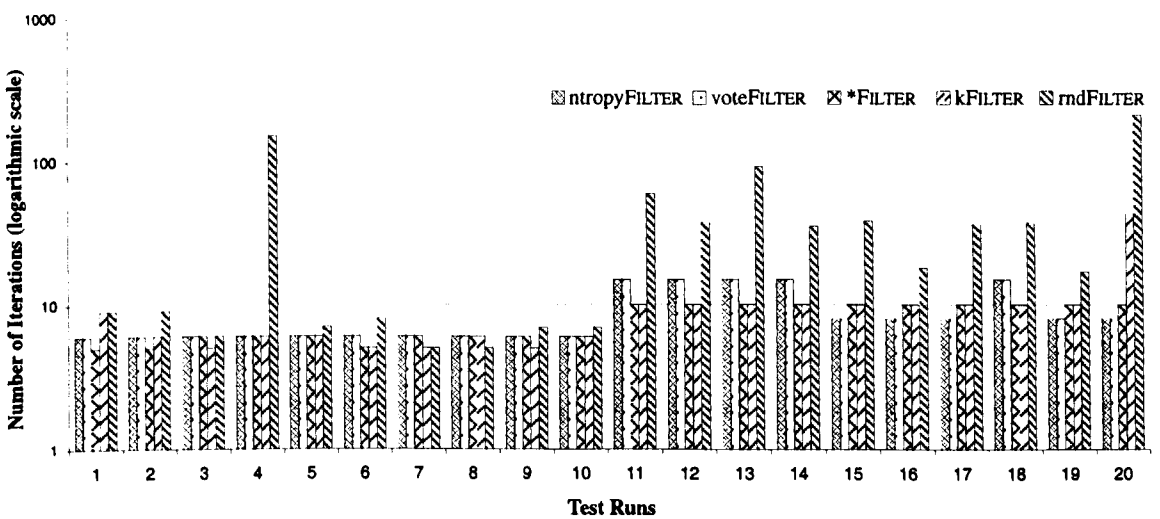


Figure 6.10: Number of Iterations for 20 Test Runs.

The number of backtracks triggered was significantly reduced by the informed selection heuristics when compared to RNDFILTER and NOFILTER (p=0.009). With increased backtracking, re-visits to previous refinement states is increased. This can have drastic effects on the number of iterations. The graph in Figure 6.10 plots the number of iterations for the 20 test runs in logarithmic scale. For instance, with the twentieth test run, the number of times backtracking is triggered with VOTEFILTER, NTROPYFILTER, *FILTER, KFILTER and RNDFILTER is 0, 0, 1, 28 and 69, while the corresponding number of iterations is 8, 8, 10, 43 and 208. Close examination of individual test runs reveals that the number of refined KBSs that pass the consistency filter can sometimes be in excess of 30. This means for RNDFILTER in the worst case, the best refined KBS will be selected only after re-visiting the example 29 times. What is interesting in this domain is that there was no significant difference in the number of backtracks triggered between *FILTER and the ensemble-based techniques. Unlike the student loans domain here, ensemble-based techniques fared well, because differences between generated ensemble members were not localised to common problem solving areas. Consequently, the ensemble consisted of a sufficient mix of members agreeing and disagreeing about solutions for affected examples. This is always more encouraging than all members agreeing or disagreeing about affected examples. Interestingly, VOTEFILTER and NTROPYFILTER have very similar results. The average processing requirements for the cluster-based *FILTER and KFILTER are on average 45% greater than the requirements for VOTEFILTER and NTROPYFILTER. Therefore, it is reasonable to suggest that the ensemble-based selection approaches are more suited to this domain.

6.5 Conclusion

The accuracy filter ranks the proposed KBSs by accuracy on all labelled examples yet to be processed. The proposed active accuracy filter extends this idea by ranking proposed KBSs based on accuracy on relevant filter examples that are actively selected from both the labelled and unlabelled example sets. Active selection of filter examples aims to select those examples that are affected by the proposed refinements.

Experimental results show that even a purely random heuristic actively selecting from both the labelled and unlabelled sets is able to improve effectiveness and efficiency, com-

pared to a passive accuracy filter using just the labelled examples in tebuf. The more informed active selection approaches attempt to select few yet relevant filter examples, thereby balancing the need for, quality (relevant) filter examples with the quantity of unlabelled filter examples that need to be labelled. The cluster-based heuristics were able to provide refined KBSs with reduced error-rates, requiring fewer re-visits to previous refinement states. However, the high computational costs associated with clustering is an obvious drawback. The ensemble based approaches are not computationally demanding and on some domains achieved similar results to cluster-based heuristics.

Chapter 7

Evaluation

The experimental evaluation reported in this chapter investigates the combined effect of example selection and refinement search strategies on the KRUSTtool. We analyse and compare improvements in refinement effectiveness and efficiency. Experimental results from all three test domains introduced in Chapter 1 will be presented and evaluated by the:

- error rate on the final output KBS;
- number of refinement cycles; and
- percentage of examples in uebuf at the end of the test run.

Reduced error-rate is an indicator of improved effectiveness, while fewer refinement cycles suggest improved efficiency. Examples in uebuf need not be labelled, but once selected must be labelled before they are useful for refinement. Therefore, examples remaining in uebuf are a good indicator of example labelling costs. The fewer remaining examples, the greater the demand on the expert.

Five KRUSTtool variants combining backtracking, example ordering, refinement and filter example selection methods are introduced in Section 7.1. Evaluation of experimental results on Student Loans, Soybean and MMU are presented in Sections 7.2, 7.3 and 7.4, followed by chapter conclusions in Section 7.5.

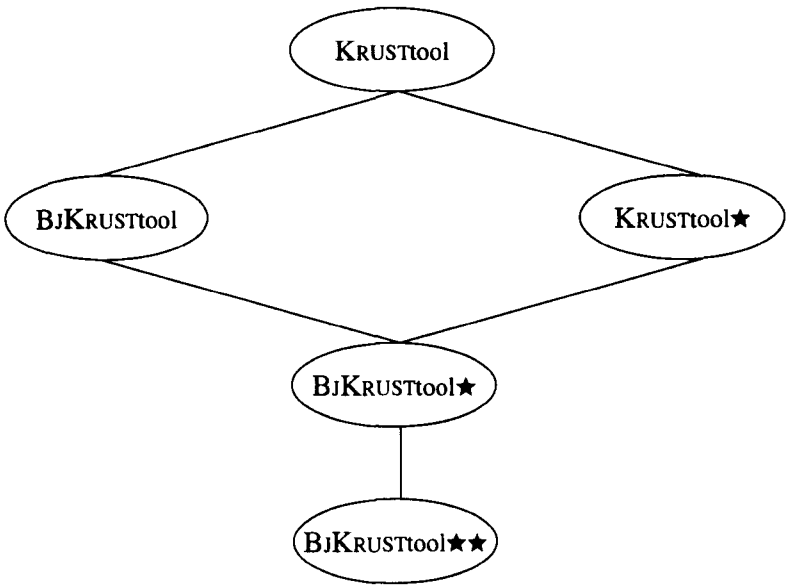


Figure 7.1: Relationship between the evaluation strategies.

7.1 KRUSTtool Variants

We compare five different combinations of example selection methods, with and without backtracking refinement search (see Figure 7.1).

KRUSTtool: without backtracking or informed selection of refinement and filter examples. Dead-ends are handled by introducing a new rule that explicitly solves the refinement example only. Refinement examples are randomly selected from uebuf, labelled and moved into tebuf. Filter examples are all remaining examples in tebuf.

KRUSTtool⋆: as KRUSTtool but with informed selection of refinement examples only.

BJKRUSTtool: as KRUSTtool but with backtracking search enabled by means of the BJ algorithm. Dead-ends will be handled by re-visiting previously abandoned refined KBSs. Static and dynamic ordering are also enabled to improve backtracking efficiency.

BJKRUSTtool⋆: as BJKRUSTtool but with informed selection of refinement examples enabled.

BJKRUSTtool⋆⋆: as BJKRUSTtool⋆, with the addition of informed selection of filter examples.

The BJ prefix indicates that backtracking search employing the BJ algorithm is enabled together with static and dynamic ordering. Suffix $\star$ indicates informed selection of refinement examples, while suffix $\star\star$ indicates informed selection of both refinement and filter examples. The KRUSTtool variants have been carefully designed with the aim of ascertaining the contribution of different example selection and utilisation strategies presented in this thesis to knowledge refinement. The experiments undertaken in this chapter will attempt to establish the following four hypothesis.

Hypothesis 1 : Accuracy of the final output KBS is significantly improved when backtracking search is enabled.

Hypothesis 2 : Number of refinement cycles is significantly reduced by resolving difficult examples in close proximity.

Hypothesis 3 : Labelling costs are significantly reduced with active selection of refinement examples without adversely affecting refinement accuracy.

Hypothesis 4 : The number of refinement dead-ends encountered during refinement search is significantly reduced with active selection of filter examples.

With BJKRUSTtool, BJKRUSTtool$\star$ and BJKRUSTtool$\star\star$, we would expect to see improved effectiveness, because dead-ends can be handled by re-starting incremental refinement from previous refinement states. Therefore improved accuracy with BJKRUSTtool, BJKRUSTtool$\star$ and BJKRUSTtool$\star\star$ compared to KRUSTtool and KRUSTtool$\star$ will establish Hypothesis 1. Hypothesis 2 relates to BJKRUSTtool$\star$'s and KRUSTtool$\star$'s ability to select difficult examples which when solved in close proximity will reduce back-jump distance resulting in fewer refinement cycles. Hypothesis 3 concerns informed selection of refinement examples as a means to reduce labelling cost by selecting few yet good examples without adversely affecting refinement accuracy. Both BJKRUSTtool$\star$ and KRUSTtool$\star$ have this facility enabled, and should at least be similar in effectiveness to their random refinement example selection counterparts BJKRUSTtool and KRUSTtool. Hypothesis 4 concerns informed selection of filter examples as a means to reduce the need to backtrack by improved direction of refinement search. Therefore with BJKRUSTtool$\star\star$, we expect comparable effectiveness to BJKRUSTtool$\star$, but achieved with fewer refinement cycles.

7.2 Student Loans Domain

Evaluation results with the Student Loans domain (Appendix A) in previous chapters, show that heuristic K-CLUSTER for selection of refinement examples, and heuristic *FILTER for selection of filter examples was best. Accordingly, BJKRUSTtool⋆, BJKRUSTtool⋆⋆ and KRUSTtool⋆, employ K-CLUSTER for informed selection of refinement examples, and BJ-KRUSTtool⋆⋆ employs *FILTER for informed selection of filter examples.

7.2.1 Error Rate

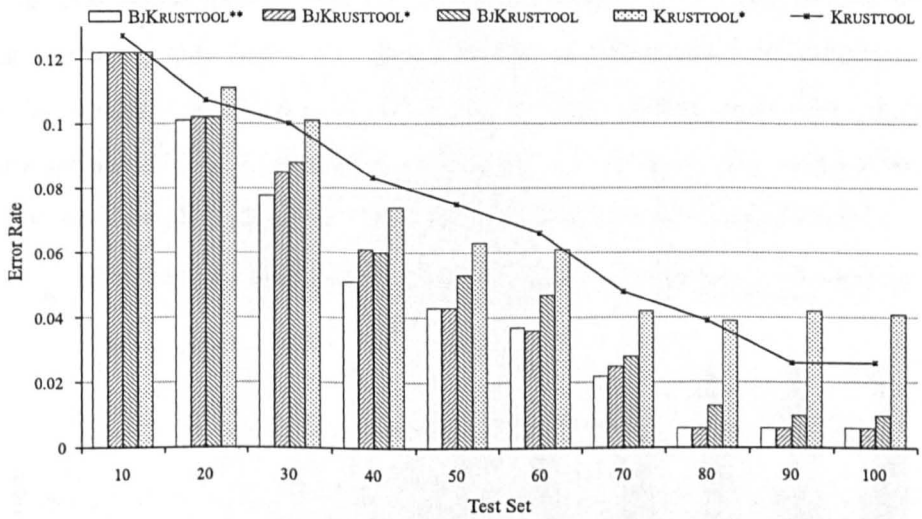


Figure 7.2: Error rate for student loans domain.

Figure 7.2 shows the averaged error rates. Statistical tests show a significant decrease in error-rate between backtracking variants over non-backtracking variants ($p = 0.002$), hence proving Hypothesis 1. Overall, BJKRUSTtool⋆⋆ has the lowest error-rates, but there was no significant difference between it and the other two backtracking variants. KRUSTtool⋆ and KRUSTtool resulted in significantly higher error rates, with greatest differences observed with test sets 80, 90 and 100. Close examination of these test sets with the backtracking variants, shows that on average backtracking was triggered in 14 of the 30 test runs. In one of these 14 test runs, 6 back-jumps were required to achieve an error-rate of 0. Therefore, it is not surprising that the non-backtracking variants performed poorly. The poor performance of KRUSTtool⋆ with these same test sets compared to KRUSTtool can be explained by KRUSTtool⋆'s inability to resolve, intelligently selected conflict pairs.

7.2.2 Refinement Cycles

Figure 7.3 shows average number of iterations. KRUSTtool has worked the least with significantly fewer refinement iterations ($p = 0.005$), but with highest error-rates. Increased number of iterations is observed with all backtracking variants. This is explained by refinement search resuming the refinement process from previous refinement states when dead-ends are encountered. BJKRUSTtool has resulted in the highest number of iterations. However, the reduced number of iterations with BJKRUSTtool$\star\star$ compared to BJKRUSTtool show that filter example selection had successfully improved refinement efficiency as postulated in Hypothesis 4. It is interesting to see that active selection of refinement examples also has a decreasing effect on the number of iterations. This is explained by heuristic K-CLUSTER's tendency to pick conflict pairs from clusters. Selected pairs get solved in relatively close proximity, requiring smaller back-jumps, thereby reducing the number of iterations. This result is consistent with hypothesis 2.

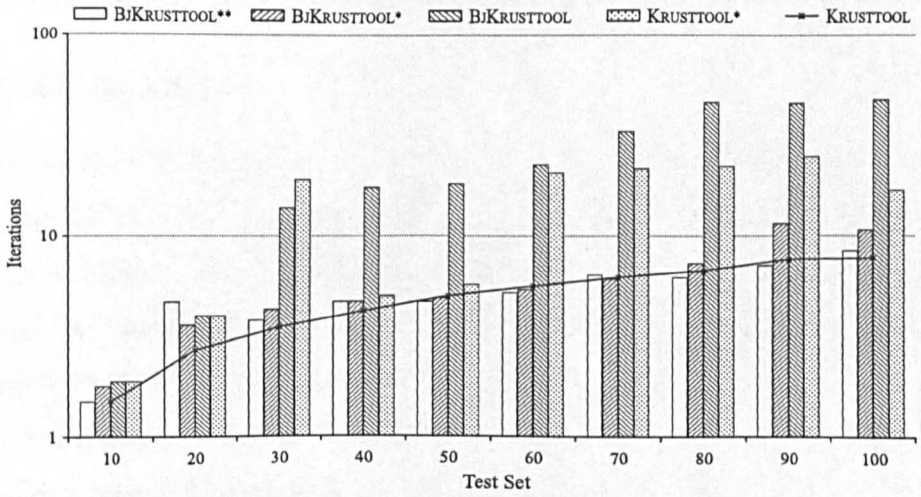


Figure 7.3: Number of iterations for student loans domain.

The significant increase in iterations with KRUSTtool$\star$ compared to KRUSTtool was unexpected. Given that the error rate of these two variants are comparable, the extra refinement effort with KRUSTtool$\star$ is clearly wasted. Close examination of 10 test runs for KRUSTtool$\star$, shows that the number of iterations would exceed 200. Actually, these runs had to be terminated once 200 was reached. Although KRUSTtool$\star$ is able to select conflict pairs, it is unable to deal with dead-ends because refinement search is disabled. Consequently, KRUSTtool$\star$ must make the correct refinement choice at each refinement

cycle, if it is to avoid dead-end situations requiring backtracking. With the 10 test runs, the choice of refined KBSs was guided only by accuracy on refinement examples. This accuracy ranking was insufficient for selecting the best refined KBS, because the number of actively selected refinement examples is far fewer when compared to the number selected randomly by KRUSTtool. Therefore, the selected non-optimal refined KBS, undid previously solved refinement examples that then went onto trigger further refinement cycles. Consequently, the same refinement examples were involved in an endless cycle of triggering refinement to no avail. Active selection of filter examples according to hypothesis 4, may offer a solution to this problem, because it aims to improve the accuracy ranking by actively selecting examples from uebuf. We tested this by evaluating KRUSTtool$\star$ on the same 10 test runs, but this time with informed selection of filter examples enabled. As expected, the number of iterations were significantly reduced to 10 on average. It seems that for iterative refinement systems that lack backtracking search, incorporating active selection of filter examples might well be the solution to overcome this deficit.

7.2.3 Labelling Effort

KRUSTtool variants employing active selection of refinement examples (suffix $\star$) have resulted in significantly higher unused percentages compared to variants without informed selection ($p = 0.001$). For instance BJKRUSTtool$\star$ has improved on BJKRUSTtool, and KRUSTtool$\star$ has improved on KRUSTtool (see Figure 7.4). These results clearly establish Hypothesis 3 where informed selection of few yet good refinement examples for labelling, reduces the demand on the expert without reducing refinement accuracy. Although BJKRUSTtool$\star\star$'s unused percentage is significantly lower than BJKRUSTtool$\star$ ($p = 0.001$), it is significantly higher than BJKRUSTtool and the rest ($p = 0.001$). The difference between BJKRUSTtool$\star\star$ and BJKRUSTtool$\star$ is to be expected due to filter example selection in BJKRUSTtool$\star\star$. Clearly there is a trade-off between reducing the number of iterations by employing active selection of filter examples, and reducing labelling effort by selecting fewer examples. Fortunately, the added cost of filter example selection and labelling pays off with improved refinement search guidance with fewer dead-ends. This suggests that, if reducing refinement effort is a priority over labelling costs then it makes sense to include active selection of filter examples and vice versa otherwise.

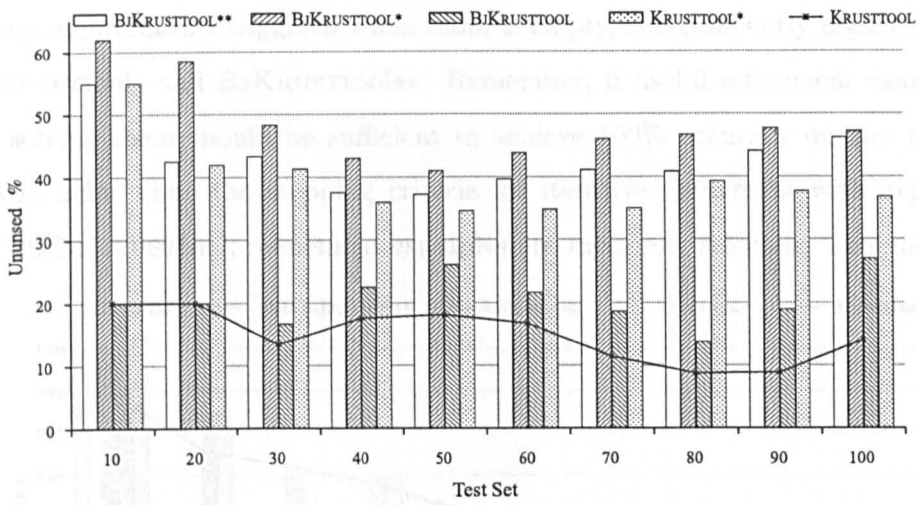


Figure 7.4: Unused example percentage for student loans domain.

7.3 Soybean Disease Domain

We apply the KRUSTtool variants to refine the Soybean KBS with 13 corruptions as discussed in Chapter 1 (Appendix B.2). Evaluation experiences with this domain in Chapters 5 and 6, show that best results were obtained with heuristic CLUSTERREP for refinement example selection and heuristic VOTEFILTER for filter example selection. Here, KRUSTtool⋆, BJKRUSTtool⋆ and BJKRUSTtool⋆⋆, apply CLUSTERREP for refinement example selection. Additionally BJKRUSTtool⋆⋆ will employ VOTEFILTER based filter example selection.

7.3.1 Error Rate

The average error-rates are shown in Figure 7.5. Both BJKRUSTtool⋆ and BJKRUSTtool⋆⋆ have significantly lower error rates compared to the rest ($p = 0.001$). This is not surprising, since a non-backtracking variant would resolve a dead-end by introducing a new rule that explicitly solves the uncorrected example only. However, with BJKRUSTtool's ability to backtrack, it is surprising that its error-rate results are similar to that of KRUSTtool and KRUSTtool⋆. The number of dead-ends encountered with BJKRUSTtool⋆⋆, BJKRUSTtool⋆ and BJKRUSTtool, provides some insight in to BJKRUSTtool's poor performance. With BJKRUSTtool⋆ and BJKRUSTtool⋆⋆ dead-ends were encountered twice on average for each of the 100 test runs. In contrast, dead-ends were never encountered with BJKRUSTtool, therefore backtracking was never actually needed. However, with BJKRUSTtool the num-

ber of sampling iterations triggered when tebuf is empty, is significantly higher compared with BJKRUSTtool⋆ and BJKRUSTtool⋆⋆. Remember, if useful refinement examples are selected, solving them should be sufficient to achieve 100% accuracy on any remaining examples in uebuf (i.e. the stopping criteria for iterative refinement with experiments here). With BJKRUSTtool, refinement examples are randomly selected. This means that

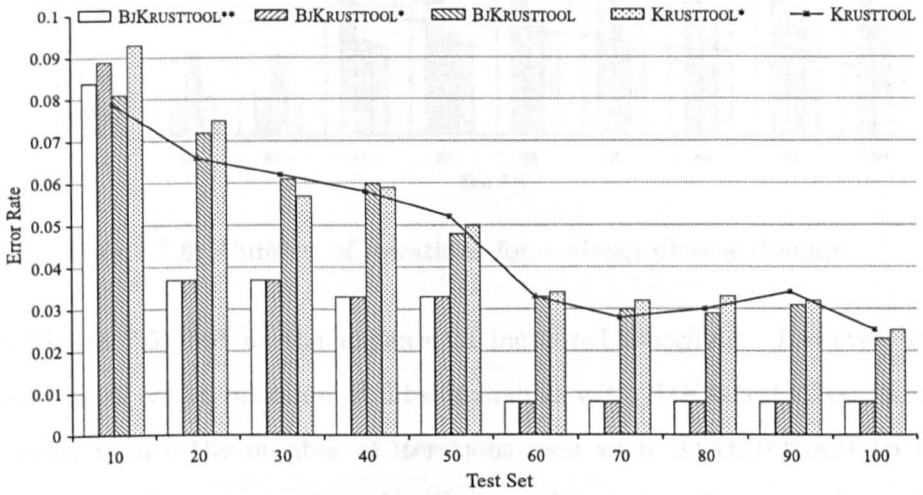


Figure 7.5: Error rate for soybean disease domain.

several sampling iterations are needed before 100% accuracy on any remaining examples in uebuf is achieved. Consequently, the number of examples selected and moved into tebuf is increased. The effect of this increase on the accuracy ranking had guided BJKRUSTtool's refinement path to a different part of the refinement search space from that of BJKRUSTtool⋆ and BJKRUSTtool⋆⋆. Unfortunately for BJKRUSTtool, that part of the search space had low accuracy on test sets even though it had 100% accuracy on training sets. The behaviour of backtracking variants in this domain is also consistent with Hypothesis 1, in that providing the opportunity to undo previous non-optimal refinements and moving refinement search to productive areas of the search space results in improved refinement accuracy.

7.3.2 Refinement Cycles

KRUSTtool⋆ and KRUSTtool have significantly fewer iterations, but they also have high error-rates. The overall trend seems to be increase in iterations leading to lower error-rates (see Figure 7.6). For instance BJKRUSTtool⋆ has achieved significantly lower error

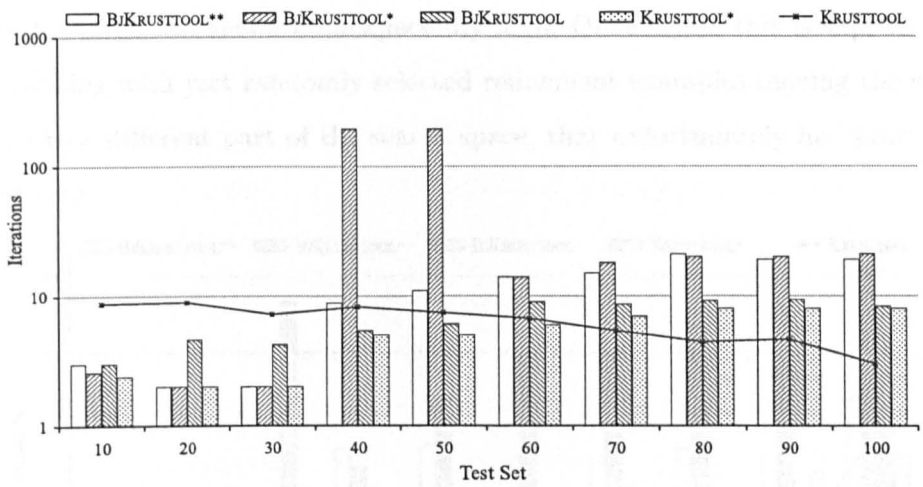


Figure 7.6: Number of iterations for soybean disease domain.

rates (see Figure 7.5) but at the expense of increased searching. However, consistent with Hypothesis 4, active selection of filter examples with BJKRUSTtool⋆⋆, has managed to significantly reduce the number of iterations seen with BJKRUSTtool⋆ ($p = 0.001$). This reduction was best demonstrated with several test runs from training sets 40 and 50. For instance, in one such test run, BJKRUSTtool⋆ consumed 195 iterations, while BJKRUSTtool⋆⋆ completed the refinement process in less than 10 iterations. Successful filtering is important, particularly when many potential refined KBSs are generated. Selection of the best refined KBS the first time round avoids needless backtracking, thereby decreasing the number of iterations. These results confirm that informed selection of filter examples helps with selecting the best refined KBS by improving the accuracy ranking.

7.3.3 Labelling Effort

BJKRUSTtool⋆ has the highest unused percentages compared with the rest as postulated in Hypothesis 3 (see Figure 7.7). Although the difference between BJKRUSTtool⋆ and BJKRUSTtool⋆⋆ is significant, both BJKRUSTtool⋆ and BJKRUSTtool⋆⋆ have significantly higher unused percentages than BJKRUSTtool, KRUSTtool⋆ and KRUSTtool. The inclusion of informed selection with and without backtracking, resulted in higher unused percentages compared to random sampling. For instance, KRUSTtool and KRUSTtool⋆ have similar error-rates, but KRUSTtool⋆ has achieved this with 20% fewer examples (on average) than KRUSTtool. Although BJKRUSTtool has 0 unused% with test sets 40 and 60, the

error-rates for these test sets are unexpectedly high. Once again, this is explained by the accuracy ranking with just randomly selected refinement examples moving the search for refinements to a different part of the search space, that unfortunately had poor accuracy on the test sets.

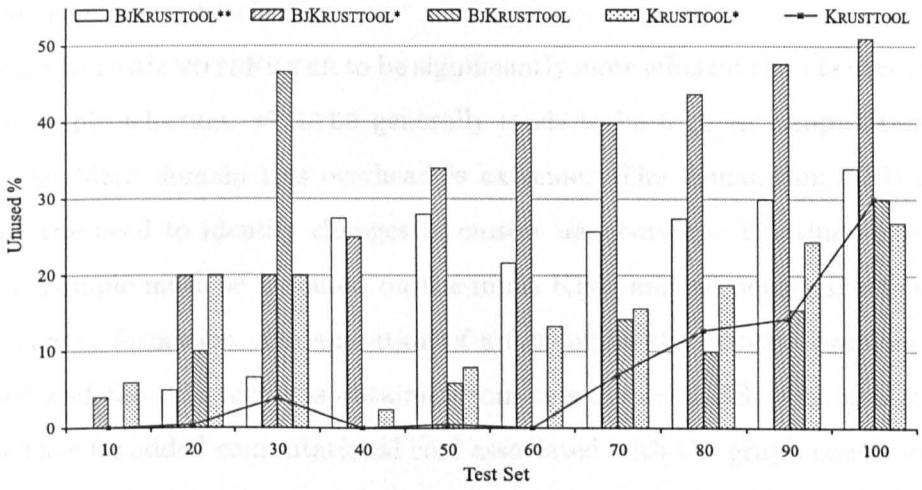


Figure 7.7: Unused example percentage for soybean disease domain.

7.4 MMU Domain

The MMU domain introduced in Chapter 1 is a real application. The original version of the MMU KBS had 2 corruptions and was used by Boswell & Craw (2000) to evaluate the effectiveness of the refinement algorithm. Here, we are interested in the effectiveness of example selection and refinement search methods, and their impact on knowledge refinement. For this purpose, differences between methods must be established using a sufficiently corrupt KBS. The MMU KBS referred to in this thesis consists of 10 additional corruptions introduced according to the available refinement operators as discussed in Chapter 1 (see Appendix C). We use 10 test runs formed according to 5x2 fold cross validation (see Section 1.5.3). Such an experimental design is able to make the most of the relatively small training set size (of 100).

A single refinement cycle with the MMU KBS can take anything from 20-45 minutes, compared with less than 5 minutes with the other 2 test domains. Therefore, any reduction in refinement iterations will have a great impact here. We found that for refinement example selection, a hybrid approach combining heuristics CLUSTERREP and *CLUSTER proved

best. Essentially, CLUSTERREP is enforced whenever intra-cluster feature dissimilarity is below a predetermined threshold and *CLUSTER is called upon otherwise. We refer to this hybrid heuristic approach as CLUSTERMIXED. Accordingly, BJKRUSTtool** and BJ-KRUSTtool*, employ heuristic CLUSTERMIXED and KRUSTtool* employs CLUSTERREP based refinement example selection.

We found heuristic VOTEFILTER to be significantly more efficient than heuristic *FILTER for filter example selection. *FILTER generally tends to be high on computational costs, and with the MMU domain this overhead is extreme. The demand on CPU resources arises from the need to identify changes in cluster membership. For this purpose, each unlabelled example must be executed on the input KBS, and all potential refined KBSs, to enable cluster formation. On execution of an example, the positive problem graph is constructed and the rule vector is obtained from this graph. KBSs with non-monotonic behaviour have an added computational cost associated with the graph construction process. This is explained by the extra backward search required to update rule activations that are retracted later in the reasoning process. Constructing a positive problem graph for a single example with the MMU KBS, takes more than 5 minutes compared to less than 20 seconds with the other 2 domains. Assuming a training set of 50 examples and 8 refined KBSs to select from, the VOTEFILTER heuristic would take at least 33.3 hours before it can begin to form example clusters, according to rule vector similarity. This delay might be reduced by forming rule vectors directly from rule traces, thereby avoiding graph construction. Of course this is not an option that scales well once rule vector similarity is extended to account for rule depths and multiple rule activations. Additionally, cluster formation for each refined KBS is also time consuming, because we now must deal with rule vectors of length 104 (compared to 20 with the student loans KBS and 44 with the soybean KBS). Therefore, a further section that compares computational overhead has been introduced with the MMU test domain, because this problem was most evident here.

7.4.1 Error Rate

Table 7.1 compares the five KRUSTtool variants on error rate. Generally similar effects were observed with nine of the 10 test runs. Only the second run was able to differentiate between the KRUSTtool variants. Here, KRUSTtool and KRUSTtool* fail to answer 5 of the 50 test examples correctly. Typically, this is explained by the lack of backtracking

search with KRUSTtool and KRUSTtool⋆ when dealing with search dead-ends. However, examination of BJKRUSTtool's refinement path, showed that backtracking was never actually needed and so was not triggered. Therefore, the differences in error rate can only be explained by the differences in refinement example selection methods. CLUSTERMIXED is employed by BJKRUSTtool⋆⋆ and BJKRUSTtool⋆ with both achieving 100% accuracy on the test set. In contrast BJKRUSTtool and KRUSTtool select examples randomly, while KRUSTtool⋆ selects according to heuristic CLUSTERREP. Actually, BJKRUSTtool⋆⋆ and BJKRUSTtool⋆ had similar refinement paths, but they both differed from the rest of the variants. Here, refinement example selection according to heuristic CLUSTERMIXED, had directed refinement search towards a path that resulted in refined KBSs with improved accuracy. The findings here expand Hypothesis 1, in that refinement accuracy can not only be improved by backtracking but also by strategic selection of refinement examples.

Test Run	BJKRUST*tool⋆⋆*	BJKRUST*tool⋆*	BJKRUST*tool*	KRUST*tool⋆*	KRUST*tool*
1	0.02	0.02	0.02	0.02	0.02
2	0.00	0.00	0.10	0.10	0.10
3	0.00	0.00	0.00	0.00	0.00
4	0.02	0.02	0.02	0.02	0.02
5	0.12	0.12	0.12	0.12	0.12
6	0.00	0.00	0.00	0.00	0.00
7	0.00	0.00	0.00	0.00	0.00
8	0.02	0.02	0.02	0.02	0.02
9	0.10	0.10	0.10	0.10	0.10
10	0.00	0.00	0.00	0.00	0.00
Avg	0.03	0.03	0.04	0.04	0.04

Table 7.1: Error rate for MMU.

Statistically the differences amongst variants with the 10 test runs are not significant. There are two contributory factors to this problem:

- the absence of a sufficiently comprehensive set of examples for experimentation purposes; and
- the difficulty of introducing controlled corruptions, which would have better enabled performance differentiation between the KRUSTtool variants.

Both problems are directly attributed to the lack of domain expertise with the complex domain of MMU, whereby manual generation of a set of examples to cover the expertise

of the KBS is difficult when the underlying reasoning is not obvious, and non-monotonic behaviour of the KBS makes it difficult to introduce meaningful corruptions.

7.4.2 Refinement Cycles

The results in Table 7.2 correspond to the number of refinement iterations with the five KRUSTtool variants. BJKRUSTtool⋆ has significantly fewer iterations compared to BJKRUSTtool, KRUSTtool⋆ and KRUSTtool (p=0.019), and therefore is consistent with Hypothesis 2. For instance, with test runs 2 and 10 KRUSTtool has 7 iterations with both test runs, compared to 4 and 3 iterations with BJKRUSTtool⋆. Here, BJKRUSTtool⋆ would have 1.5 to 2 hours time saving with these 2 test runs. However, the difference between BJKRUSTtool⋆ and BJKRUSTtool⋆⋆ is not significant. This may suggest that informed selection of refinement examples alone is adequate to direct refinement search, without any need for informed filter example selection, thereby refuting Hypothesis 4. However, remember that with both Student loans and Soybean test domains, BJKRUSTtool⋆⋆ was able to reduce the number of refinement cycles when compared to BJKRUSTtool⋆. Therefore, it is safer to deduce that it is the absence of suitable filter examples in the training set that resulted in an insignificant difference between BJKRUSTtool⋆ and BJKRUSTtool⋆⋆, rather than the non-optimal performance of the filter example selection heuristic itself.

Test Run	BJKRUST*tool*⋆⋆	BJKRUST*tool*⋆	BJKRUST*tool*	KRUST*tool*⋆	KRUST*tool*
1	5	4	4	6	5
2	4	4	5	5	7
3	4	4	5	5	5
4	4	4	5	4	6
5	3	4	3	3	4
6	4	4	4	6	6
7	3	4	5	5	5
8	4	4	4	5	4
9	3	3	3	3	5
10	3	3	5	5	7
Avg	3.7	3.8	4.3	4.7	5.4

Table 7.2: Number of iterations for MMU.

7.4.3 Labelling Effort

Table 7.3 shows the unused example percentage results for each of the KRUSTtool variants. Informed selection of refinement examples should enable BJKRUSTtool⋆ and KRUSTtool⋆ to have the highest unused percentage results. Since active filter example selection is enabled with BJKRUSTtool⋆⋆, we would expect unused percentage results that are not as high as BJKRUSTtool⋆ and KRUSTtool⋆, but still significantly higher than KRUSTtool and BJKRUSTtool. Statistical results confirm these expectations postulated in Hypothesis 3, where BJKRUSTtool⋆ and KRUSTtool⋆ have significantly higher unused percentages compared to BJKRUSTtool and KRUSTtool ($p = 0.001$), and also compared to BJKRUSTtool⋆⋆ ($p = 0.001$). These results are most obvious with the second test run. Here, BJKRUSTtool⋆ has used 46% of the examples in uebuf, while BJKRUSTtool has used all examples, and KRUSTtool has used 49 of the 50 examples. With some test runs, KRUSTtool⋆ has slightly higher unused percentages over BJKRUSTtool⋆. This is explained by the different heuristics that are employed for refinement example selection. CLUSTERREP employed by KRUSTtool⋆, tends to select fewer examples from clusters compared to the hybrid heuristic CLUSTERMIXED employed by BJKRUSTtool⋆. BJKRUSTtool⋆⋆ has used up more examples than BJKRUSTtool⋆, but this is explained by the additional selection of filter examples. In the real world where expert interaction is often limited and labelling costs are high, informed selection of training examples, be it for refinement or filtering purposes, will be a valuable asset.

Test Run	BJKRUST*tool⋆⋆*	BJKRUST*tool⋆*	BJKRUST*tool*	KRUST*tool⋆*	KRUST*tool*
1	36	52	32	52	32
2	32	46	0	44	2
3	44	38	60	54	32
4	46	52	40	54	40
5	38	48	20	42	20
6	44	52	40	50	40
7	40	48	40	52	40
8	36	48	40	48	40
9	46	48	20	54	20
10	34	48	20	54	24
Avg	39.6	48	31.2	50.4	29

Table 7.3: Unused example percentage for MMU.

7.4.4 Computational Overhead

Computational costs increase with KBS complexity and non-monotonic reasoning adds to this complexity. With real applications, complex KBSs are to be expected and savings in computational costs will be an advantage. Analysis of example selection and refinement search methods, clearly show that example clustering is most demanding on CPU resources. There are two ways in which the clustering process can be improved to reduce this overhead:

- improving the efficiency of the clustering algorithm; and
- reducing the number of times examples need be clustered.

Some of the issues related to improving algorithm efficiency will be discussed under future work in Chapter 8. Here, we concentrate on reducing the number of clustering episodes. With informed selection of refinement examples, we would either select a random example from each cluster, or select k examples from the cluster with highest intra-feature dissimilarity. Such an approach can be wasteful because information that can be derived by both heuristics about example clusters is not being exploited. With CLUSTERMIXED we have combined the two heuristics and hope to achieve improved selection efficiency. More importantly we expect that such a hybrid selection approach will also increase the number of useful examples that can be selected from a single clustering episode.

Table 7.4 compares the number of sampling iterations, CPU cycles and unused example percentages for three versions of BJKRUSTtool⋆. The results for BJKRUSTtool⋆ with heuristic CLUSTERREP is on the left, with K-CLUSTER in the center, and CLUSTERMIXED at the right. It is clear that with increased number of sampling iterations the number of CPU cycles will also increase dramatically. Remember that each sampling iteration involves a single clustering episode, therefore reducing the number of times examples need to be clustered will reduce computational costs. CLUSTERMIXED has significantly fewer sampling iterations compared with the rest ($p = 0.01$). The contradictory results with test runs 3 and 9, where CLUSTERMIXED has one extra sampling iteration over CLUSTERREP can only be explained by CLUSTERREP's random selection of an example from each cluster. Essentially, the randomly selected cluster representatives happened to be better with these test runs. The unused example percentage tends to be slightly higher with some

BJKRUSTtool⋆								
CLUSTERREP			K-CLUSTER			CLUSTERMIXED		
Sampl. Iter.	*CPU cycles*	*Unused %*	*Sampl. Iter.*	*CPU cycles*	*Unused %*	*Sampl. Iter.*	*CPU cycles*	*Unused %*
2	5005160	52	2	6180340	**70**	1	4613560	52
2	6996610	44	3	8483480	24	1	4631800	46
1	3790960	**54**	5	6621190	**2**	**2**	4006730	**38**
2	**6309190**	**54**	3	6641990	16	1	**4948020**	**52**
2	4157000	42	2	3808990	50	1	3040700	48
2	5960690	50	3	8266330	26	1	4518140	52
2	**5472400**	**52**	4	8537800	22	1	**4571100**	**48**
1	4550840	48	3	5569040	28	1	4551930	48
1	2409600	**54**	2	3871160	12	**2**	2597100	**48**
1	**3640080**	**54**	4	8366440	10	1	**3168630**	**48**
8	4829253	50.4	3.1	6634676	26	1.2	4064771	48

Table 7.4: Comparing computational costs with refinement example selection heuristics CLUSTERREP, K-CLUSTER and CLUSTERMIXED

test runs for CLUSTERREP. However, if reducing computational costs is also a priority, then CLUSTERMIXED presents itself as a balanced choice. Results with K-CLUSTER are somewhat erratic. For instance, the number of sampling iterations ranges from 2 to 5, and unused percentages from 70 to 2. This clearly suggests that a hybrid approach is better suited to this domain, whereby CLUSTERREP's general selection approach is complimented with K-CLUSTER's localised selection approach.

7.5 Conclusion

A consistent observation is that improved accuracy is achieved when backtracking is enabled, thus establishing Hypothesis 1. Of course this is in addition to the availability of refinement examples that can expose faults in the KBS. However, improved accuracy with backtracking is achieved at the expense of increased refinement iterations. Endless looping is a more severe problem that occurs, particularly when backtracking is not enabled. There is evidence to suggest that filter example selection can help avoid looping by directing refinement search to more promising parts of the search space as postulated in Hypothesis 4.

The experimental results suggests that improvement in accuracy can be achieved with a small but representative set of refinement and filter examples. This is explained by

the influence of selected examples on the accuracy ranking. Essentially, in the absence of a representative set of filter examples, the accuracy ranking can adversely influence refined KBS selection, such that incremental refinement is moved to areas of the refinement search space that have a detrimental effect in general. This is consistent with Hypothesis 4, which proposed informed selection of filter examples as a mechanism to improve refinement efficiency in general. A more obvious advantage of informed example selection is the reduction in labelling costs demonstrated with all three test domains, hence proving Hypothesis 3.

Computational cost is an unavoidable issue when dealing with real world applications. The hybrid approach to refinement example selection addresses this problem with encouraging results. Additionally, the choice of VOTEFILTER over *FILTER for filter example selection with the MMU domain highlights the need to mix and match heuristics to suit the application domain.

We have recommended different example selection heuristics for the three test domains in this chapter. The diversity of available example selection heuristics necessitates some guidance regarding the selection of appropriate heuristics. Although there are no obvious answers for a real setting it is possible to recommend a hybrid selection heuristic such as CLUSTERMIXED for refinement example selection because such a heuristic will address KBSs both, with and without interacting faults. For filter example selection the VOTEFILTER selection approach is advised when the execution of KBSs involves high computational costs while the *FILTER selection approach is suitable for KBSs with interacting faults and low execution costs.

Chapter 8

Conclusion

The research work reported in this thesis was undertaken as part of the KRUSTWorks project. A KRUSTtool is a KBS specific refinement tool, assembled from the KRUSTWorks generic refinement toolkit. The KRUSTtool's approach to knowledge refinement is iterative, where the refinement algorithm attempts to fix one or more, but typically not all, of the wrongly-solved examples in the training set. It is also incremental because the output KBS selected from a set of potential refined KBSs, becomes the input KBS in the next iteration. This iterative incremental approach to knowledge refinement can be viewed as a search task; a search for the best refined KBS through the space of possible refinements. Accordingly, the proposed solutions in this thesis are two-fold, considering:

- training example utilisation strategies to improve *refinement search*, by incorporating backtracking to previous refinement states and enforcing an order on the sequence of repairs; and
- *informed selection* of training examples to drive and guide refinement search, with particular emphasis on reducing the demand on expert labelling costs.

The search and selection strategies proposed are novel, and exploit techniques from unsupervised learning; ensemble based learning; and constraint satisfaction search. They have been built into the generic KRUSTWorks framework. However, the strategies are sufficiently general that they are applicable to any iterative refinement tool that adopts an incremental approach to refinement and is able to capture the problem solving behaviour of the KBS.

Section 8.1 examines issues related with refinement search. Example selection strategies according to the refinement and filtering role of examples are discussed in Section 8.2. General conclusions from experimentation with refinement search and example selection strategies are presented in Section 8.3. The contributions of this thesis are outlined in Section 8.4, followed by desirable extensions in Section 8.5, and a summary in Section 8.6.

8.1 Refinement Search

The refinement task is sufficiently complex that the space of possible repairs demands a heuristic search, typically hill-climbing. EITHER (Ourston & Mooney 1994) and FORTE (Richards & Mooney 1995), try to repair the outstanding fault that is indicated by the *largest* number of examples, and choose the repair with the *fewest* changes, to rules which are *nearest* the observables. A KRUSTtool's refinement algorithm also applies hill-climbing search. Although it generates many refined KBSs designed to fix each incorrect example, it then chooses the refined KBS with the *highest* accuracy on training examples (those yet to be processed) as the input KBS for the next iteration of the algorithm. The result is that refinement tools are dogged by the standard hill-climbing problem of getting caught in local optima. The problem can be solved by re-starting refinement search from a previously abandoned refinement state whenever a local optimum is detected. A local optimum is reached when all generated refined KBSs are unable to improve refinement accuracy. Such situations are common, because one or more previously solved examples can get undone by all generated refined KBSs. The undoing of previously solved examples with iterative knowledge refinement draws close parallels to the undoing of previously instantiated variables with constraint satisfaction search.

CSP search reaches a dead-end when a variable cannot be instantiated because of inconsistencies with previously instantiated variables. The solution involves undoing previously instantiated variables and re-starting the process from a previous solution state. With iterative refinement the hill climbing search can be converted into a best first search, that is willing to commit to previously abandoned paths whenever dead-ends are encountered by incorporating backtracking CSP search strategies. CSP search strategies vary in the manner in which the re-starting point is ascertained. Experiments presented in Chapter 3 show that the combination of iterative refinement with the BJ search strat-

egy resulted in refined KBSs that had significantly lower error-rates. However, this was achieved at the expense of increased number of refinement cycles, suggesting a need for improved efficiency.

The analogy between CSP search and knowledge refinement is taken a step further by examining various search ordering heuristics that are employed by CSPs, which may provide some insight as to how the efficiency of iterative knowledge refinement can be improved. Variable and value ordering heuristics (Dechter & Meiri 1994, Gent, MacIntyre, Prosser, Smith & Walsh 1996), help identify variables that are most constrained so that these can be dealt with first. Invariably there is a need for estimation of variable constrainedness, and for identifying the sources from which this constrainedness information is to be derived.

For knowledge refinement, the problem solving behaviour of the rule-base and fault evidence from training examples yet to be processed, were good information sources for estimating training example constrainedness. Various static (Tsang 1993) and dynamic (Haralick & Elliott 1980) ordering schemes were implemented using heuristics that exploit this constrainedness information. Experimental results in Chapter 4 indicate that the decision to employ static or dynamic ordering schemes must be made keeping in mind the trade-off between cpu-usage and refinement cycles. The accuracy of the final KBS was however, not significantly affected in any way.

8.2 Informed Selection

Refinement examples are those training examples with which the refinement cycle is triggered. These examples are labelled based on the expert's solutions for a given range of problem tasks. If expert interaction is limited, it is important that we select few yet good training examples for labelling. In contrast, if there are many labelled examples available for refinement then, given that the refinement process is quite computationally expensive, it is convenient to select those examples whose repairs also fix other wrongly solved examples without further refinement, thereby reducing the number of refinement cycles.

Agglomerative clustering techniques were employed to identify a subset of *good* training examples for knowledge refinement. The training examples are clustered based on the area of the rule-base being exercised. For instance, all examples triggering similar rules

are more likely to be clustered together, i.e. the similarity metric captures the problem solving behaviour of the rule-base, with respect to the training examples. Various selection heuristics are then employed to select one or more examples from each of these clusters.

Refinement filtering is the final stage in a refinement cycle, where the best refined KBS is selected from a subset of potential refined KBSs as the output KBS for the next iteration. It is an important stage, because *good* selection criteria will reduce the need to backtrack to previous refinement states. This means reduced refinement iterations and considerable savings on computational costs. The accuracy ranking phase of filtering was the primary area of interest, since it involves ranking based on accuracy on a selected subset of training examples, referred to as *filter examples*. This ranking can be adversely affected when it is based on:

- a non-representative set of examples, particularly consisting of examples unaffected by the potential refined KBSs; or
- a large set of examples consuming considerable computation resources.

Filter example selection methods proposed in this thesis, aim to identify examples affected by potential refined KBSs. The accuracy ranking itself is then based on a representative subset of examples, selected from those that are identified as affected. An ensemble based approach selects examples that are solved most differently by the ensemble. This involves a measure of disagreement between the ensemble members, reflecting consensus about how the example was solved. Conveniently, the ensemble constitutes the set of potential refined KBSs from which the best is to be selected. A different approach exploits and extends the clustering framework for selection of filter examples in addition to refinement examples. Here, the strategy involves several clustering episodes, where each potential refined KBS will have a corresponding example clustering. Affected examples can then be identified by analysing changes to cluster membership between clusters formed for each potential refined KBS and the input KBS. The changes can be difficult to track and this is tackled by considering only those changes that affect the cluster from which the refinement example was selected. Although a cluster based selection of filter examples proved very effective, the high computational costs makes it impractical for real applications.

8.3 Experimental Results

Evaluation has been a continuous process throughout this project. This is clearly reflected by experimental results presented at the end of most chapters. However, a thorough evaluation on two artificial and one real domain was undertaken with the objective of analysing the affects of different example selection methods, refinement ordering methods and refinement search methods, separately and with respect to each other. For this purpose the experiments were designed to investigate the isolated and combined effects of refinement search and example selection strategies on the KRUSTtool. Five KRUSTtool variants combining backtracking with example ordering, active refinement and filter example selection methods were analysed.

The results suggest that backtracking variants have significantly improved accuracy over the non-backtracking ones. When active selection of refinement examples is enabled, the KRUSTtool generates refined KBSs with similar accuracy using fewer examples. However, it was interesting to observe that a hybrid selection method was necessary to achieve similar results with the real application domain. Filter example selection when enabled, effectively guides the KRUSTtool through the refinement space, thereby reducing the need to backtrack to previous refinement states. Overall informed selection of examples, suggests that improved accuracy can be achieved with fewer labelled examples. In the real world this would mean effective use of a busy expert's time.

8.4 Main Contributions

The work reported in this thesis falls under research pertaining to knowledge refinement, particularly example selection and utilisation for iterative knowledge refinement of rule-based systems. Therefore, the primary contributions are to knowledge refinement research and these are:

- resolution of the problem of refinement dead-ends by the conversion of the hill-climbing best first search into one that is able to backtrack to previous refinement stages, thereby improving refinement accuracy;
- improved refinement search efficiency by incorporating heuristics that enforce example ordering so that related examples are dealt close together;

- an example clustering framework employing a rule-vector based similarity metric, enabling cluster formation reflective of the underlying problem solving behaviour of the KBS;
- heuristics that aim to select few yet good examples from these clusters, thereby reducing labelling costs;
- extension of the clustering framework as a mechanism to identify filter examples by monitoring changes in cluster membership;
- an ensemble-based approach to filter example selection, where potential refined KBSs form the ensemble, and filter examples are those examples that cause the greatest disagreement amongst the ensemble members; and
- a comprehensive evaluation of all refinement search and example selection methods on three test domains.

Knowledge refinement is incremental learning where the learning must adapt existing knowledge in the KBS with the aid of training examples. Essentially, knowledge refinement falls under the broader context of machine learning. Invariably work in this thesis also contributes to machine learning in general, as a novel approach to active selection of examples for iterative knowledge refinement using unsupervised learning. The use of rule vector based similarity for this purpose is an interesting idea that can be exploited with active selection of examples for machine learning algorithms. One possibility is to perform several induction episodes with different features assigned as the concept to be learned (because at sampling we do not have example labels). The rule vectors for examples can then be formed on the basis of all induced rule sets.

Incorporating CSP search strategies within the knowledge refinement framework has been an interesting and challenging experience. Like most experiences there are useful lessons to be learned and here it is the need for a variety of dynamic CSP strategies. It is apparent that CSP solving methods must increasingly be adapted to cope with complex applications involving both dynamic variables and dynamic values. Undoubtedly this will broaden the horizon for CSP methods in the real world.

8.5 Desirable Extensions

The thesis opens new possibilities for selective sampling applied to machine learning algorithms and for further improvements and extensions to example selection and refinement search methods for iterative knowledge refinement.

8.5.1 Backtracking Search

Presently, refinement search can fail when extensive backtracking regresses refinement to the root (start of the refinement path). Typically, such a situation arises when refinement conflicts resolved by inducing a new antecedent, a new value, or a new rule, was not successful. Successful operation of induction operators depends not only on the availability of examples, but also on the ability to select a representative subset of training examples for induction. Often the examples that trigger backtracking and are in conflict with previously solved refinement examples, provide a good source of examples for induction. In the unfortunate event of regressing to the start of the refinement path, all refinement examples involved in backtracking could be exploited for induction purposes in the next refinement search attempt.

Often when backtracking employing the BJ algorithm, a new refinement path is initiated from a previous state by skipping over several refinement states. Typically the skipped over states do not contribute to the refinement conflict that triggered backtracking. Consequently, exploring the refinement path leads to re-discovery, of these skipped over states. With CSP search strategies the *sticky values* heuristic is employed to avoid re-discovery by remembering the current value of a variable while skipping over it during a back-jump (Frost & Dechter 1994, Kambhampati 2000). The underlying intuition is that skipping over a variable means that its instantiated value did not contribute to the conflict that triggered a back-jump. Therefore, on re-visiting this variable, the remembered value is restored. A further variation enforces value ordering with the remembered value at the front and any before it appended to the end. What might be of interest to knowledge refinement is that remembering potential repairs and consolidating these may help reduce computational costs, particularly with real applications. However, the task complexity of incremental refinement compared to variable instantiation with CSP, suggests a need for far more complex mechanisms that enable merging of remembered refinements with differ-

ent input KBSs, when refinement examples are re-visited. Nevertheless, it is an interesting opportunity for reducing computational costs.

8.5.2 Refinement Example Ordering

The ordering of refinement examples can directly control the sequence of refinements. Ideally, refinement examples that expose interacting faults in the KBS need to be resolved in close proximity during the early stages of refinement search. Identifying these examples amounts to identifying examples that enforce the greatest constraint on the choice of possible refinements. The ordering heuristics presented in chapter 4 obtain information about refinement constrainedness in two ways: from the problem graph; and fault evidence from training examples yet to be processed. With both forms, ties between examples with similar constrainedness are broken randomly. However, it is easy to see how a hybrid heuristic might be employed where the problem graph heuristics can be augmented with the fault evidence based heuristics to resolve ties (or vice versa). Such an approach would be similar to the Brelaz heuristic (Brelaz 1979), which combines variable and value ordering for CSPs.

8.5.3 Example Selection Efficiency

Experimental results in Chapter 7 show that highest demands on CPU resources are associated with problem graph creation and pair-wise distance calculations during cluster formation. An improved clustering algorithm using a two-stage approach is employed by McCallum, Nigam & Ungar (2000) with encouraging results when applied to a 1916 sized dataset. The two-stage approach involves:

1. an initial cheap partition of examples into overlapping subsets; followed by
2. an expensive clustering step that needs only to calculate pair-wise distances between examples in overlap regions.

The cheap partitions are referred to as *canopies*. Essentially a canopy contains a subset of examples that according to the cheap similarity measure is within a pre determined distance threshold. Of course this requires some understanding of the underlying properties of the examples that are to be clustered before suitable thresholds can be set in place.

However, the advantage of this approach is evident if the initial cheap distance measure is able to form canopies such that there exists a canopy for every cluster. Here, the more expensive distance measure need only be applied to the overlap areas forming distinct clusters from overlapping canopies.

Figure 8.1 shows five clusters and the canopies that cover them. Examples belonging to the same cluster are coloured in the same shade of grey. Each canopy is initiated from a randomly selected example (or seed). Here starting with canopy A, further canopies can be formed from examples placed outside A's dotted line (or inner threshold). Expensive distance measurements will only be made between examples sharing a canopy. Essentially, the distance between examples not sharing a canopy can be set to infinity.

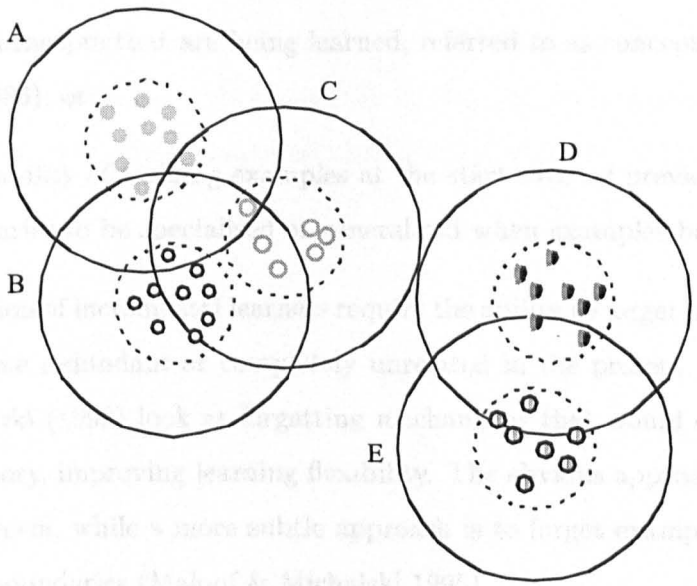


Figure 8.1: Clusters covered by overlapping canopies.

A cheap distance metric for example clustering with knowledge refinement might be rule vector formation according to a *subset* of rule activations. This subset might be formed by considering only rule activations with n-depth from the bottom (with data driven reasoning) or the top (with goal driven reasoning) of the problem graph. Such an approach would improve computational efficiency by:

- reducing the number of *complete* positive problem graphs that need be created; and
- reducing pair wise similarity calculation costs due to smaller rule vectors.

Canopies formed using the smaller rule vectors can then be refined by applying hierarchical clustering with complete rule vectors and expensive pair-wise distances, calculated for only those examples that fall under two or more canopies.

8.5.4 Selective Memory Retention

Incremental learning involves continuous iterations of learning a concept according to a set of available training examples at a given time, and the possible adaptation of this concept to cover newer training examples as they are made available. Incremental learners in a real world situation must typically evolve according to changes in the real world. The reasons for this can be:

- changes to concepts that are being learned, referred to as concept drift (Schlimmer & Fisher 1986); or
- the unavailability of training examples at the start causing previously learned concept boundaries to be specialised or generalised when examples become available.

Successful operation of incremental learners require the ability to forget past examples that have either become redundant or completely unrelated in the present. For this purpose Maloof & Michalski (1999) look at forgetting mechanisms that would efficiently manage the learners memory, improving learning flexibility. The obvious approach is to introduce an ageing mechanism, while a more subtle approach is to forget examples that no longer enforce concept boundaries (Maloof & Michalski 1995).

Iterative knowledge refinement can benefit from a forgetting mechanism, particularly when operating in environments where concept drift is the norm. This would entail an ageing mechanism over cebuf with the effect of reduced back-jumps to previously solved examples that are undone but are now redundant because of concept drift. A more direct benefit of a forgetting mechanism is that it may help reduce computational costs, because fewer examples means fewer pair-wise similarity calculations for clustering. Of course this would hold only if the cost of implementing a forgetting mechanism is sufficiently lower than the cost of clustering without these mechanisms in place.

8.6 Thesis Summary

This thesis has explored training example selection and utilization methods, with the aim of improving the efficiency and effectiveness of knowledge refinement. Iterative incremental knowledge refinement is a search for the best refined KBS through the space of possible refinements. Refinement search leading to non-optimal parts of the refinement space can result in refinement dead-ends which should ideally be dealt with backtracking search. In the absence of backtracking search, informed filtering mechanisms when in place, ensure refinement search is directed to parts of the search space with improved refinement accuracy. If example clustering is used as the basis for informed selection of examples, it then makes sense to employ a similarity metric that reflects the underlying task of the examples. Finally, improved refinement accuracy can be achieved with few yet good refinement examples. The challenge however, is to identify the best mix of refinement search and example selection strategies that improve effectiveness and efficiency for a given application domain.

Appendix A

Corrupted Student Loans Rule-base in Clips

```
;-----------------------------------------------
; CLIPS Student Loans Adviser KBS
; Corruptions to Rules R6, E1, R16, R17 and R19
; -----------------------------------------------

(defrule R1
   (continuously_enrolled ?Student)
   =>
   (assert (no_payment_due ?Student)))

(defrule R2
   (eligible_for_deferment ?Student)
   =>
   (assert (no_payment_due ?Student)))

(defrule R3
   (declare (salience -20))
   (not (no_payment_due ?Student))
   =>
   (assert (payment_due ?Student)))

(defrule R4
   (absence ?Student ?days)
   (test (> 6 ?days))
```

```
   =>
   (assert (never_left_school ?Student)))

(defrule R5
   (enrolled ?Student ?School ?Units)
   (school ?School)
   =>
   (assert (enrolled_in_n_units ?Student ?Units ?School)))

(defrule R6
   (never_left_school ?Student)
   (enrolled_in_n_units ?Student ?units ?School)
   (test (>= ?units 5))  ;; CORRUPTED: should be > 5
   =>
   (assert (continuously_enrolled ?Student)))

(defrule R8
   (military_deferment ?Student)
   =>
   (assert (eligible_for_deferment ?Student)))

(defrule R9
   (peace_corps_deferment ?Student)
   =>
   (assert (eligible_for_deferment ?Student)))

(defrule R10
   (financial_deferment ?Student)
   =>
   (assert (eligible_for_deferment ?Student)))

(defrule R11
   (student_deferment ?Student)
   =>
   (assert (eligible_for_deferment ?Student)))

(defrule R12
   (disability_deferment ?Student)
   =>
```

```
   (assert (eligible_for_deferment ?Student)))

(defrule R14
   (enlisted ?Student ?org)
   (armed_forces ?org)
   =>
   (assert (military_deferment ?Student)))

(defrule R15
   (enlisted ?Student ?org)
   (peace_corps ?org)
   =>
   (assert (peace_corps_deferment ?Student)))

(defrule R16
   (filed_for_bankruptcy ?Student yes)
   (enlisted ?Student ?)  ;; CORRUPTED: extra condition
   =>
   (assert (financial_deferment ?Student)))

(defrule R17
   (unemployed ?Student yes)
   (enrolled_in_n_units ?Student 20 ?School) ;; CORRUPTED: extra condition
   =>
   (assert (financial_deferment ?Student)))

(defrule E1                                  ;; CORRUPTED: extra rule
   (enrolled ?Student uci ?units)
   =>
   (assert (financial_deferment ?Student)))

(defrule R18
   (enrolled_in_n_units ?Student ?units ?School)
   (test (> ?units 11))
   =>
   (assert (student_deferment ?Student)))

(defrule R19
   (disabled ?Student yes)
```

```
(filed_for_bankruptcy ?Student yes) ;; CORRUPTED: extra condition
=>
(assert (disability_deferment ?Student)))
```

Appendix B

Corrupted Soybean Rule-base in Clips

Two corrupted Soybean KBSs are included below. The first KBS has 7 corruptions and the second has 13 corruptions.

B.1 Soybean Corrupted Rule-base I

```
;-----------------------------------------------------
; CLIPS Soybean Disease Diagnosis KBS
; Corruptions to Rules R1, R2, R16, R7, R14, R20 and E1
; -----------------------------------------------------

(defrule R1
    (fruiting_bodies present ?plant) ;; CORRUPTED: extra condition
    (plant_stand normal ?plant)
    (int_discolor none ?plant)
    (seed norm ?plant)
    (diaporthe_stem_canker_condition ?plant)
    =>
    (assert (diagnosis diaporthe-stem-canker ?plant)))

(defrule R2
    (plant_growth abnorm ?plant) ;; CORRUPTED extra condition
    (int_discolor black ?plant)
=>
```

```
    (assert (diagnosis charcoal-rot ?plant)))

(defrule R16
    (int_discolor black ?plant) ;; CORRUPTED extra condition
    (leaves norm ?plant)
    (int_discolor none ?plant)
    (rhizoctonia_root_rot_condition ?plant)
    =>
    (assert (diagnosis rhizoctonia-root-rot ?plant)))

(defrule R17
    (int_discolor black ?plant)
    (plant_stand normal ?plant)
    (low_temp ?plant)
    (stem abnorm ?plant)
    (int_discolor none ?plant)
    (rhizoctonia_root_rot_condition ?plant)
    =>
    (assert (diagnosis rhizoctonia-root-rot ?plant)))

(defrule R3
    (plant_growth abnorm ?plant)
    =>
    (assert (rhizoctonia_root_rot_condition ?plant)))

(defrule R18
    (int_discolor none ?plant)
    (roots rotted ?plant)
    (phytophthora_root_rot_condition ?plant)
    =>
    (assert (diagnosis phytophthora-root-rot ?plant)))

(defrule R19
    (plant_stand <_normal ?plant)
    (area_damaged low_areas ?plant)
    (fruiting_bodies absent ?plant)
    (phytophthora_root_rot_condition ?plant)
    =>
    (assert (diagnosis phytophthora-root-rot ?plant)))
```

```
(defrule E1                          ;; CORRUPTED extra rule
    (hail yes ?plant)
    (stem_cankers above_sec_nde ?plant)
    =>
    (assert (hail_canker_relation ?plant)))

(defrule R4
    (plant_growth abnorm ?plant)
    (leaves abnorm ?plant)
    =>
    (assert (phytophthora_root_rot_condition ?plant)))

(defrule R5
    (leaves abnorm ?plant)
    (stem abnorm ?plant)
    (lodging yes ?plant)
    (int_discolor brown ?plant)
    =>
    (assert (diagnosis brown-stem-rot ?plant)))

(defrule R50
    (leafspot_size dna ?plant)
    (int_discolor none ?plant)
    =>
    (assert (powdery_mildew_condition ?plant)))

(defrule R6
    (powdery_mildew_condition ?plant)
    (leaves abnorm ?plant)
    (stem_cankeronds absent ?plant)
    =>
    (assert (diagnosis powdery-mildew ?plant)))

(defrule R7
    (area_damaged whole_field ?plant) ;; CORRUPTED extra condition
    (leafspot_size >_1/8 ?plant)
    (mold_growth present ?plant)
```

```
    =>
    (assert (diagnosis downy-mildew ?plant)))

(defrule R9
    (leafspots_marg w-s_marg ?plant)
    (leafspot_size <_1/8 ?plant)
    (canker_lesion dna ?plant)
    (seed_size norm ?plant)
    =>
    (assert (diagnosis bacterial-blight ?plant)))

(defrule R14
    (season ?d ?plant)
    (test (>= ?d 5))  ;; CORRUPTED 5 is incorrect should be 8
    (test (<= ?d 10))
    (leafspot_size >_1/8 ?plant)
    (leafspots_halo no_yellow_halos ?plant)
    (leaves abnorm ?plant)
    (leafspots_marg w-s_marg ?plant)
    (leaf_shread absent ?plant)
    =>
    (assert (alternarialeaf_spot_condition ?plant)))

(defrule R15
    (leafspot_size >_1/8 ?plant)
    =>
    (assert (frog_eye_leaf_spot_condition ?plant)))

(defrule R20
    (leafspot_size >_1/8 ?plant) ;; CORRUPTED extra condition
    (plant_growth norm ?plant)
    (fruiting_bodies present ?plant)
    (fruit_pods norm ?plant)
    =>
    (assert (diagnosis brown-spot ?plant)))

(defrule R21
    (season 6 ?plant)
    (precip >_norm ?plant)
```

```
    (leafspot_size >_1/8 ?plant)
    (mold_growth absent ?plant)
    =>
    (assert (diagnosis brown-spot ?plant)))

(defrule R22
    (season 5 ?plant)
    (leafspot_size >_1/8 ?plant)
    (leaf_malf absent ?plant)
    (mold_growth absent ?plant)
    =>
    (assert (diagnosis brown-spot ?plant)))

(defrule R23
    (season 7 ?plant)
    (area_damaged whole_field ?plant)
    (leafspot_size >_1/8 ?plant)
    (leaf_malf absent ?plant)
    (fruit_pods norm ?plant)
    =>
    (assert (diagnosis brown-spot ?plant)))

(defrule R24
    (season 4 ?plant)
    (stem norm ?plant)
    =>
    (assert (diagnosis brown-spot ?plant)))

(defrule R25
    (leafspots_marg no_w-s_marg ?plant)
    (int_discolor none ?plant)
    =>
    (assert (diagnosis bacterial-pustule ?plant)))

(defrule R26
    (leafspot_size <_1/8 ?plant)
    (seed_size <_norm ?plant)
    =>
    (assert (diagnosis bacterial-pustule ?plant)))
```

```
(defrule R27
    (leafspot_size <_1/8 ?plant)
    (canker_lesion tan ?plant)
    =>
    (assert (diagnosis purple-seed-stain ?plant)))

(defrule R28
    (leaves norm ?plant)
    (leafspot_size dna ?plant)
    (stem_cankers absent ?plant)
    (int_discolor none ?plant)
    =>
    (assert (diagnosis purple-seed-stain ?plant)))

(defrule R29
    (plant_growth norm ?plant)
    (stem_cankers above_sec_nde ?plant)
    (leafspot_size dna ?plant)
    =>
    (assert (diagnosis anthracnose ?plant)))

(defrule R30
    (plant_growth abnorm ?plant)
    (stem abnorm ?plant)
    (seed abnorm ?plant)
    =>
    (assert (diagnosis anthracnose ?plant)))

(defrule R31
    (plant_stand <_normal ?plant)
    (plant_growth abnorm ?plant)
    (fruiting_bodies present ?plant)
    =>
    (assert (diagnosis anthracnose ?plant)))

(defrule R32
    (leafspot_size >_1/8 ?plant)
    (leaf_malf present ?plant)
```

```
    (mold_growth absent ?plant)
    =>
    (assert (diagnosis phyllosticta-leaf-spot ?plant)))

(defrule R33
    (precip <_norm ?plant)
    (leafspot_size >_1/8 ?plant)
    (int_discolor none ?plant)
    =>
    (assert (diagnosis phyllosticta-leaf-spot ?plant)))

(defrule R34
    (season 7 ?plant)
    (precip norm ?plant)
    (leafspot_size >_1/8 ?plant)
    =>
    (assert (diagnosis phyllosticta-leaf-spot ?plant)))

(defrule R35
    (int_discolor none ?plant)
    (fruit_pods norm ?plant)
    (mold_growth absent ?plant)
    (alternarialeaf_spot_condition ?plant)
    =>
    (assert (diagnosis alternarialeaf-spot ?plant)))

(defrule R36
    (stem norm ?plant)
    (mold_growth absent ?plant)
    (alternarialeaf_spot_condition ?plant)
    =>
    (assert (diagnosis alternarialeaf-spot ?plant)))

(defrule R37
    (hail yes ?plant)
    (plant_growth norm ?plant)
    (leaf_malf absent ?plant)
    (fruiting_bodies absent ?plant)
    (fruit_pods norm ?plant)
```

```
    (mold_growth absent ?plant)
    (alternarialeaf_spot_condition ?plant)
    =>
    (assert (diagnosis alternarialeaf-spot ?plant)))

(defrule R38
    (plant_growth norm ?plant)
    (stem abnorm ?plant)
    (fruiting_bodies absent ?plant)
    (frog_eye_leaf_spot_condition ?plant)
    =>
    (assert (diagnosis frog-eye-leaf-spot ?plant)))

(defrule R39
    (fruit_pods diseased ?plant)
     =>
    (assert (diagnosis frog-eye-leaf-spot ?plant)))

(defrule R40
    (season 8 ?plant)
    (plant_stand normal ?plant)
    (seed_tmt fungicide ?plant)
    (leaf_shread absent ?plant)
    (fruiting_bodies absent ?plant)
    (mold_growth absent ?plant)
    (frog_eye_leaf_spot_condition ?plant)
     =>
    (assert (diagnosis frog-eye-leaf-spot ?plant)))

(defrule R41
    (season 8 ?plant)
    (hail no ?plant)
    (fruiting_bodies absent ?plant)
    (seed norm ?plant)
    (frog_eye_leaf_spot_condition ?plant)
     =>
    (assert (diagnosis frog-eye-leaf-spot ?plant)))

(defrule R42
```

```
    (season 7 ?plant)
    (precip >_norm ?plant)
    (area_damaged scattered ?plant)
    (mold_growth absent ?plant)
    (frog_eye_leaf_spot_condition ?plant)
    =>
    (assert (diagnosis frog-eye-leaf-spot ?plant)))

(defrule R43
    (leafspots_halo no_yellow_halos ?plant)
    =>
    (assert (leafspots_halo ?plant)))

(defrule R44
    (leafspots_halo yellow_halos ?plant)
    =>
    (assert (leafspots_halo ?plant)))

(defrule R45
    (temp norm ?plant) => (assert (high_temp ?plant)))

(defrule R46
    (temp >_norm ?plant) => (assert (high_temp ?plant)))

(defrule R47
    (temp norm ?plant) => (assert (low_temp ?plant)))

(defrule R48
    (temp <_norm ?plant) => (assert (low_temp ?plant)))

(defrule R51
    (temp norm ?plant)
    (plant_growth abnorm ?plant)
    (stem abnorm ?plant)
    =>
    (assert (diaporthe_stem_canker_condition ?plant)))
```

B.2 Soybean Corrupted Rule-base II

```
;-----------------------------------------------------
; CLIPS Soybean Disease Diagnosis KBS
; Corruptions to Rules R1, R2, R16, R7, R14, R20, R21,
; R22, R34, E1, E2, E3 and E4
; -----------------------------------------------------

(defrule R1
    (fruiting_bodies present ?plant) ;; CORRUPTED: extra condition
    (plant_stand normal ?plant)
    (int_discolor none ?plant)
    (seed norm ?plant)
    (diaporthe_stem_canker_condition ?plant)
    =>
    (assert (diagnosis diaporthe-stem-canker ?plant)))

(defrule R2
    (plant_growth abnorm ?plant) ;; CORRUPTED extra condition
    (int_discolor black ?plant)
=>
    (assert (diagnosis charcoal-rot ?plant)))

(defrule R16
    (int_discolor black ?plant) ;; CORRUPTED extra condition
    (leaves norm ?plant)
    (int_discolor none ?plant)
    (rhizoctonia_root_rot_condition ?plant)
    =>
    (assert (diagnosis rhizoctonia-root-rot ?plant)))

(defrule R17
    (int_discolor black ?plant)
    (plant_stand normal ?plant)
    (low_temp ?plant)
    (stem abnorm ?plant)
    (int_discolor none ?plant)
    (rhizoctonia_root_rot_condition ?plant)
    =>
```

```
    (assert (diagnosis rhizoctonia-root-rot ?plant)))

(defrule R3
    (plant_growth abnorm ?plant)
    =>
    (assert (rhizoctonia_root_rot_condition ?plant)))

(defrule E1 ;; CORRUPTED extra rule
    (rhizoctonia_root_rot_condition ?plant)
    =>
    (assert (diagnosis rhizoctonia-root-rot ?plant)))

(defrule R18
    (int_discolor none ?plant)
    (roots rotted ?plant)
    (phytophthora_root_rot_condition ?plant)
    =>
    (assert (diagnosis phytophthora-root-rot ?plant)))

(defrule R19
    (plant_stand <_normal ?plant)
    (area_damaged low_areas ?plant)
    (fruiting_bodies absent ?plant)
    (phytophthora_root_rot_condition ?plant)
    =>
    (assert (diagnosis phytophthora-root-rot ?plant)))

(defrule R4
    (plant_growth abnorm ?plant)
    (leaves abnorm ?plant)
    =>
    (assert (phytophthora_root_rot_condition ?plant)))

(defrule R5
    (leaves abnorm ?plant)
    (stem abnorm ?plant)
    (lodging yes ?plant)
    (int_discolor brown ?plant)
    =>
```

```
    (assert (diagnosis brown-stem-rot ?plant)))

(defrule R50
    (leafspot_size dna ?plant)
    (int_discolor none ?plant)
    =>
    (assert (powdery_mildew_condition ?plant)))

(defrule R6
    (powdery_mildew_condition ?plant)
    (leaves abnorm ?plant)
    (stem_cankeronds absent ?plant)
    =>
    (assert (diagnosis powdery-mildew ?plant)))

(defrule R7
    (area_damaged whole_field ?plant) ;; CORRUPTED extra condition
    (leafspot_size >_1/8 ?plant)
    (mold_growth present ?plant)
    =>
    (assert (diagnosis downy-mildew ?plant)))

(defrule R9
    (leafspots_marg w-s_marg ?plant)
    (leafspot_size <_1/8 ?plant)
    (canker_lesion dna ?plant)
    (seed_size norm ?plant)
    =>
    (assert (diagnosis bacterial-blight ?plant)))

(defrule R14
    (season ?d ?plant)
    (test (>= ?d 5))  ;; CORRUPTED 5 is incorrect should be 8
    (test (<= ?d 10))
    (leafspot_size >_1/8 ?plant)
    (leafspots_halo no_yellow_halos ?plant)
    (leaves abnorm ?plant)
    (leafspots_marg w-s_marg ?plant)
    (leaf_shread absent ?plant)
```

```
    =>
    (assert (alternarialeaf_spot_condition ?plant)))

(defrule E2
    (season ?d ?plant)
    (test (>= ?d 3))
    =>
    (assert (alternarialeaf_spot_condition ?plant)))

(defrule R15
    (leafspot_size >_1/8 ?plant)
    =>
    (assert (frog_eye_leaf_spot_condition ?plant)))

(defrule R20
    (leafspot_size >_1/8 ?plant) ;; CORRUPTED extra condition
    (plant_growth norm ?plant)
    (fruiting_bodies present ?plant)
    (fruit_pods norm ?plant)
    =>
    (assert (diagnosis brown-spot ?plant)))

(defrule R21
    (season ?d ?plant)
    (test (> ?d 6))  ;; CORRUPTED incorrect operator should be =
    (precip >_norm ?plant)
    (leafspot_size >_1/8 ?plant)
    (mold_growth absent ?plant)
    =>
    (assert (diagnosis brown-spot ?plant)))

(defrule R22
    (season 5 ?plant)
    (leafspot_size >_1/8 ?plant)
    (fruiting_bodies present ?plant) ;; CORRUPTED extra condition
    (leaf_malf absent ?plant)
    (mold_growth absent ?plant)
    =>
    (assert (diagnosis brown-spot ?plant)))
```

```
(defrule R23
    (season 7 ?plant)
    (area_damaged whole_field ?plant)
    (leafspot_size >_1/8 ?plant)
    (leaf_malf absent ?plant)
    (fruit_pods norm ?plant)
    =>
    (assert (diagnosis brown-spot ?plant)))

(defrule R24
    (season 4 ?plant)
    (stem norm ?plant)
    =>
    (assert (diagnosis brown-spot ?plant)))

(defrule R25
    (leafspots_marg no_w-s_marg ?plant)
    (int_discolor none ?plant)
    =>
    (assert (diagnosis bacterial-pustule ?plant)))

(defrule R26
    (leafspot_size <_1/8 ?plant)
    (seed_size <_norm ?plant)
    =>
    (assert (diagnosis bacterial-pustule ?plant)))

(defrule E3  ;; CORRUPTED extra rule
    (leafspot_size <_1/8 ?plant)
    =>
    (assert (sig-diagnosis bacterial-pustule ?plant)))

(defrule R27
    (leafspot_size <_1/8 ?plant)
    (canker_lesion tan ?plant)
    =>
    (assert (diagnosis purple-seed-stain ?plant)))
```

```
(defrule R28
    (leaves norm ?plant)
    (leafspot_size dna ?plant)
    (stem_cankers absent ?plant)
    (int_discolor none ?plant)
    =>
    (assert (diagnosis purple-seed-stain ?plant)))

(defrule R29
    (plant_growth norm ?plant)
    (stem_cankers above_sec_nde ?plant)
    (leafspot_size dna ?plant)
    =>
    (assert (diagnosis anthracnose ?plant)))

(defrule R30
    (plant_growth abnorm ?plant)
    (stem abnorm ?plant)
    (seed abnorm ?plant)
    =>
    (assert (diagnosis anthracnose ?plant)))

(defrule R31
    (plant_stand <_normal ?plant)
    (plant_growth abnorm ?plant)
    (fruiting_bodies present ?plant)
    =>
    (assert (diagnosis anthracnose ?plant)))

(defrule R32
    (leafspot_size >_1/8 ?plant)
    (leaf_malf present ?plant)
    (mold_growth absent ?plant)
    =>
    (assert (diagnosis phyllosticta-leaf-spot ?plant)))

(defrule R33
    (precip <_norm ?plant)
    (leafspot_size >_1/8 ?plant)
```

```
    (int_discolor none ?plant)
    =>
    (assert (diagnosis phyllosticta-leaf-spot ?plant)))

(defrule R34
    (season ?d ?plant)
    (test (>= ?d 4)) ;; CORRUPTED incorrect value should be 7
    (test (<= ?d 7))
    (precip norm ?plant)
    (leafspot_size >_1/8 ?plant)
    =>
    (assert (diagnosis phyllosticta-leaf-spot ?plant)))

(defrule R35
    (int_discolor none ?plant)
    (fruit_pods norm ?plant)
    (mold_growth absent ?plant)
    (alternarialeaf_spot_condition ?plant)
    =>
    (assert (diagnosis alternarialeaf-spot ?plant)))

(defrule R36
    (stem norm ?plant)
    (mold_growth absent ?plant)
    (alternarialeaf_spot_condition ?plant)
    =>
    (assert (diagnosis alternarialeaf-spot ?plant)))

(defrule R37
    (hail yes ?plant)
    (plant_growth norm ?plant)
    (leaf_malf absent ?plant)
    (fruiting_bodies absent ?plant)
    (fruit_pods norm ?plant)
    (mold_growth absent ?plant)
    (alternarialeaf_spot_condition ?plant)
    =>
    (assert (diagnosis alternarialeaf-spot ?plant)))
```

```
(defrule R38
    (plant_growth norm ?plant)
    (stem abnorm ?plant)
    (fruiting_bodies absent ?plant)
    (frog_eye_leaf_spot_condition ?plant)
    =>
    (assert (diagnosis frog-eye-leaf-spot ?plant)))

(defrule R39
    (fruit_pods diseased ?plant)
     =>
    (assert (diagnosis frog-eye-leaf-spot ?plant)))

(defrule R40
    (season ?d ?plant)
    (test (>= ?d 7))
    (test (<= ?d 10)) ;; CORRUPTED incorrect value should be 8
    (plant_stand normal ?plant)
    (seed_tmt fungicide ?plant)
    (leaf_shread absent ?plant)
    (fruiting_bodies absent ?plant)
    (mold_growth absent ?plant)
    (frog_eye_leaf_spot_condition ?plant)
     =>
    (assert (diagnosis frog-eye-leaf-spot ?plant)))

(defrule R41
    (season 8 ?plant)
    (hail no ?plant)
    (fruiting_bodies absent ?plant)
    (seed norm ?plant)
    (frog_eye_leaf_spot_condition ?plant)
     =>
    (assert (diagnosis frog-eye-leaf-spot ?plant)))

(defrule R42
    (season 7 ?plant)
    (precip >_norm ?plant)
    (area_damaged scattered ?plant)
```

```
    (mold_growth absent ?plant)
    (frog_eye_leaf_spot_condition ?plant)
    =>
    (assert (diagnosis frog-eye-leaf-spot ?plant)))

(defrule R43
    (leafspots_halo no_yellow_halos ?plant)
    =>
    (assert (leafspots_halo ?plant)))

(defrule R44
    (leafspots_halo yellow_halos ?plant)
    =>
    (assert (leafspots_halo ?plant)))

(defrule R45
    (temp norm ?plant) => (assert (high_temp ?plant)))

(defrule R46
    (temp >_norm ?plant) => (assert (high_temp ?plant)))

(defrule R47
    (temp norm ?plant) => (assert (low_temp ?plant)))

(defrule R48
    (temp <_norm ?plant) => (assert (low_temp ?plant)))

(defrule E4                          ;; CORRUPTED extra rule
    (hail yes ?plant)
    (stem_cankers above_sec_nde ?plant)
    =>
    (assert (hail_canker_relation ?plant)))

(defrule R51
    (temp norm ?plant)
    (plant_growth abnorm ?plant)
    (stem abnorm ?plant)
    =>
    (assert (diaporthe_stem_canker_condition ?plant)))
```

Appendix C

Corrupted MMU Rule-base in Clips

The size of the corrupted MMU rule-base makes it inpractical to be appended in full. Therefore, only the 12 corrupted individual rules are listed below.

```
;-----------------------------------------------
; The corrupted parts of the
; CLIPS Manned Maneuvering Unit KBS (MMU KBS)
; -----------------------------------------------

;pos x input
(defrule cea-a-test-input-posx-null-null-1
        ;;  CORRUPTED (side c on) added to disjunction
        (or (aah off) (side c on) (and (gyro on)
                                       (gyro movement none none)))
        (side a on)
        (side b on)
        (rhc roll none pitch none yaw none)
        (thc x pos y none z none)
        (or
        (vda a f2 off)
        (vda a f3 off)
        (vda a ?n&~f1&~f2&~f3&~f4 on)
        )
=>
        (assert (failure cea))
        (assert (suspect a))
        (printout t "failure -during translational command" crlf)
```

```
        (printout t " in the pos x direction" crlf)
        (assert (conclusion cea translational pos x))
)

(defrule cea-b-test-input-posx-null-null-1
        (or (aah off) (and (gyro on) (gyro movement none none)))
        (side a on)
        (side b on)
        (thc x pos y none z pos)     ;CORRUPTED extra condition
        (rhc roll none pitch none yaw none)
        (thc x pos y none z none)
        (or
        (vda b f1 off)
        (vda b f4 off)
        (vda b ?n&~f1&~f2&~f3&~f4 on)
        )
=>
        (assert (failure cea))
        (assert (suspect b))
        (printout t "failure -during translational command " crlf)
        (printout t "in the pos x direction" crlf)
        (assert (conclusion cea translational pos x))
)

;pos z,
(defrule cea-a-test-input-posz-null-null-11
        (or (aah off) (and (gyro on) (gyro movement none none)))
        (side a on)
        (side b on)
        (thc x pos y none z neg)     ;CORRUPTED extra condition
        (rhc roll none pitch none yaw none)
        (thc x none y none z pos)
        (or
        (vda a d1 off)
        (vda a d2 off)
        (vda a ?n&~d1&~d2 on)
        )
=>
        (assert (failure cea))
```

```
        (assert (suspect a))
        (printout t "failure -during translational command " crlf)
        (printout t " in the pos z direction" crlf)
        (assert (conclusion cea translational pos z))
)

(defrule cea-b-test-input-posz-null-null-11
        (or (aah off) (and (gyro on) (gyro movement none none)))
        (side a on)
        (side b on)
        (rhc roll none pitch none yaw none)
        (thc x pos y none z neg)     ;CORRUPTED extra condition
        (thc x none y none z pos)
        (or
        (vda b d1 off)
        (vda b d2 off)
        (vda b ?n&~d1&~d2 on)
        )
=>
        (assert (failure cea))
        (assert (suspect b))
        (printout t "failure -during translational command " crlf)
        (printout t ";; in the pos z direction" crlf)
        (assert (conclusion cea translational pos z))
)

(defrule cea-b-test-input-neg-null-null-12
        (or (aah off) (and (gyro on) (gyro movement none none)))
        (side a on)
        (side b on)
        (thc x pos y none z pos)     ; CORRUPTED extra condition
        (rhc roll none pitch none yaw none)
        (thc x none y none z neg)
        (or
        (vda b u3 off)
        (vda b u4 off)
        (vda b ?n&~u3&~u4 on)
        )
=>
```

```
        (assert (failure cea))
        (assert (suspect b))
        (printout t crlf ";; failure during translational command " crlf )
        (printout t ";; in the neg z direction" crlf)
        (assert (conclusion cea translational pos z))
)

;pos x
(defrule cea-test-input-pos-null-null-side-a-1
        ;;  CORRUPTED added (side c on) to disjunction
        (or (aah off) (and  (side c on) (gyro on)
                            (gyro movement none none)))
        (not (checking thrusters))
        (side a on)
        (side b off)
        (rhc roll none pitch none yaw none)
        (thc x pos y none z none)
        (or
        (vda a f2 off)
        (vda a f3 off)
        (vda a ?n&~f2&~f3 on)
        )
=>
        (assert (failure cea))
        (assert (suspect a))
        (printout t crlf ";; cea failure on side a" crlf)
        (assert (conclusion cea failure side-a))
)

;pos pitch
(defrule cea-test-input-null-pos-null-side-a-3
        (or (aah off) (and (gyro on)(gyro movement none none)))
        (not (checking thrusters))
        (side a on)
        (side b off)
        (side d)      ; CORRUPTED extra condition
        (rhc roll none pitch pos yaw none)
        (thc x none y none z none)
        (or
```

```
        (vda a b1 off)
        (vda a f3 off)
        (vda a ?n&~b1&~f3 on)
        )
=>
        (assert (failure cea))
        (assert (suspect a))
        (printout t crlf ";; cea failure on side a" crlf )
        (assert (conclusion cea failure side-a))
)

;neg x
(defrule cea-test-input-neg-null-null-side-b-2
        (or (aah off) (and (gyro on) (gyro movement none none)))
        (not (checking thrusters))
        (side a off)
        (side b on)
        (rhc roll none pitch neg yaw pos) ;; CORRUPTED extra condition
        (rhc roll none pitch none yaw none)
        (thc x neg y none z none)
        (or
        (vda b b2 off)
        (vda b b3 off)
        (vda b ?n&~b2&~b3 on)
        )
=>
        (assert (failure cea))
        (assert (suspect b))
        (printout t crlf ";; cea failure on side b" crlf )
        (assert (conclusion cea failure side-b))
)

;neg pitch
(defrule cea-test-input-null-neg-null-side-b-4
        (or (aah off) (and (gyro on)(gyro movement none none)))
        (not (checking thrusters))
        (side a off)
        (side b on)
        (rhc roll none pitch neg yaw pos) ;CORRUPTED extra condition
```

```
        (rhc roll none pitch neg yaw none)
        (thc x none y none z none)
        (or
        (vda b f1 off)
        (vda b b3 off)
        (vda b ?n&~f1&~b3 on)
        )
=>
        (assert (failure cea))
        (assert (suspect b))
        (printout t crlf ";; cea failure on side b" crlf )
        (assert (conclusion cea failure side-b))
)

(defrule no-xfeed-fuel-reading-test-side-a-grt
        (declare (salience -10))
        (xfeed-a closed)
        (xfeed-b closed)
        (not (failure ?))
        (not (side b on))      ; CORRUPTED incorrect negation
        (fuel-used-a ?fuel-a)
        (tank-pressure-was a ?p-old)
        (tank-pressure-current a ?p-new)
        (test (< (- ?p-old ?fuel-a) ?p-new))
        ?x <- (side a on)
        (side b on)
=>
        (assert (failure thruster-a))
        (printout t crlf ";; pressure in tank a is high,
                          a thruster has not responded"crlf)
        (printout t crlf ";; side a failed while executing
                          thruster commands" crlf)
        (assert (conclusion thruster-a high-pressure))
        (assert (conclusion cea failure side-a))
        (assert (side a off))
        (retract ?x)
        (assert (checking thrusters))
)
```

```
(defrule no-xfeed-fuel-reading-test-side-b-grt-1
        (declare (salience -10))
        (xfeed-a closed)
        (xfeed-b closed)
        (side c)          ; CORRUPTED extra condition
        (not (failure ?))
        (fuel-used-b ?fuel-b)
        (tank-pressure-was b ?p-old)
        (tank-pressure-current b ?p-new)
        (test (< (- ?p-old ?fuel-b) ?p-new))
        (side a on)
        ?x <- (side b on)
=>
        (assert (failure thruster-b))
        (printout t crlf ";; pressure in tank b is high,
                          a thruster has not responded"crlf)
        (printout t crlf ";; side b failed while executing thruster
        commands" crlf)
        (assert (conclusion thruster-b high-pressure))
        (assert (conclusion cea failure side-b))
        (assert (side b off))
        (retract ?x)
        (assert (checking thrusters))
)

(defrule no-xfeed-fuel-reading-test-side-b-grt-2
        (declare (salience -10))
        (xfeed-a closed)
        (xfeed-b closed)
        (not (failure ?))
        (not (side b on))      ; CORRUPTED incorrect negation
        (fuel-used-b ?fuel-b)
        (tank-pressure-was b ?p-old)
        (tank-pressure-current b ?p-new)
        (test (> (- ?p-old ?fuel-b) ?p-new))
        (side a on)
        ?x <- (side b on)
=>
        (assert (failure thruster-b))
```

```
        (printout t crlf ";; pressure in tank b is low, " crlf)
        (printout t ";; possible uncommanded acceleration
                       or fuel leak" crlf crlf)
        (printout t crlf ";; side b failed while
                         executing thruster commands" crlf)
        (assert (conclusion thruster-b low-pressure))
        (assert (conclusion cea failure side-b))
        (assert (side b off))
        (retract ?x)
        (assert (checking thrusters))
)
```

Appendix D

Interpretation of Results

We found that most of our evaluation results were not normally distributed. In these situations non-parametric statistical tests are better suited as they do not require the assumption of normality, and use the median instead of the average as the basis of comparison (Mendenhall & Sincich 1988, Anderson, Sweeney & Williams 1990). Additionally, non-parametric tests tend to be more robust than their parametric counterparts. Two commonly used non-parametric tests are:

- the Wilcoxen signed rank test for comparison of two data sets; and
- the Kruskal-Wallis test for comparison of two or more data sets.

With the Wilcoxen signed rank test we have a choice between the matched-pairs test or the independent random samples test, depending on whether or not the 2 data samples comprise of matched-pairs. The Wilcoxen statistic is the number of pairwise averages that are greater than the comparison value plus one half the number equal to the comparison value. Typically, for the matched-pairs test the comparison value is 0, and the null hypothesis is that the median equals this comparison value and the alternative hypothesis is that it is greater (or less) than the comparison value. With unmatched-pairs the null hypothesis is that the median of the first sample is greater (or less) than the second samples median, and the alternative hypothesis is that they are equal. The Wilcoxen statistic is interpreted according to the p value, where a p value less than 0.05 indicates that the probability of rejecting the null hypothesis and being wrong is less than 0.05.

The Kruskal-Wallis test ranks the data sets to be compared. It consists of the H statistic that can be interpreted according to the p-value, where a p value less than 0.05 indicates a 95% significant difference between one or more of the data sets that are being compared. In MINITAB (Minitab-Inc. 1998) the z value can then be used to identify which of the data sets are ranked significantly above or below the group's median rank. For a two-tailed test the z value is significant at the 95% confidence level if > 1.96 or < -1.96. The sign of the z value indicates whether the difference is greater (plus) or less (minus) than the group's median rank.

Appendix E

Published Papers

- Wiratunga and Craw (1999a). Incorporating Backtracking in Knowledge Refinement, *Proceedings of the 5th European Symposium on Verification and Validation of Knowledge Based Systems and Components (EUROVAV99)*, Kluwer, Oslo, Norway, pp 193-205.

- Wiratunga and Craw (1999b). Sequencing Training Examples for Iterative Knowledge Refinement, *Proceedings of the 19th SGES International Conference on Knowledge Based Systems and Applied Artificial Intelligence (ES99)*, Springer, Cambridge, UK, pp 41-56.

- Wiratunga and Craw (2000). Informed selection of Training Examples for Knowledge Refinement, *Proceedings of the 12th International Conference on Knowledge Engineering and Knowledge Management (EKAW2000)*, Springer, Juan-les Pins, France, pp 233-248.

Bibliography

Anderson, D. A., Sweeney, D. J. & Williams, T. A. (1990). *Statistics for Business and Economics*, West Publishing Company, St. Paul, MN.

Angluin, D. (1988). Queries and concept learning, *Machine Learning* **2**: 319–342.

Angluin, D., Frazier, M. & Pitt, L. (1992). Learning conjunctions of horn clauses, *Machine Learning* **9**: 147–164.

Argamon-Engelson, S. & Dagan, I. (1999). Committee-based sample selection for probabilistic classifiers, *Journal of Artificial Intelligence Research* **11**: 335–360.

Ayel, M. & Vignollet, L. (1993). SYCOJET and SACCO, two tools for verifying expert systems, *International Journal of Expert Systems* **6**(3): 357–382.

Bart Selman, H. K. & Cohen, B. (1994). Noise strategies for improving local search, *Proceedings of the Twelfth National Conference on Artificial Intelligence*, Seattle, WA.

Bartak, R. (1999). Constraint programming - what is behind?, *In Proceedings of the 1st Workshop on Constraint Programming for Decision and Control (Invited Talk)*, Gliwice, Poland.

Bartak, R. (2000). Dynamic constraint models for planning and scheduling problems, *Proceedings of ERCIM Working Group on Constraints/Compulog Net Area on Constraint Programming.* to appear.

Bitner, J. R. & Reingold, E. (1975). Backtrack programming techniques, *Communications of the ACM* **18**: 651–656.

Blake, C., Keogh, E. & Merz, C. (1998). UCI repository of machine learning databases. http://www.ics.uci.edu/~mlearn/MLRepository.html.

Blum, A. L. & Langley, P. (1997). Selection of relevant features and examples in machine learning, *Artificial Intelligence* **1-2**: 245–271.

Boswell, R. & Craw, S. (1999). Organising Knowledge Refinement Operators, *Validation and Verification of Knowledge Based Systems, Proceedings of the 5th European Symposium on the Validation and Verification of Knowledge Based Systems (EUROVAV'99)*, Kluwer, Oslo, Norway, pp. 149–161.

Boswell, R. & Craw, S. (2000). Experiences with a generic refinement toolkit, *EKAW2000*, Springer Verlag. to appear.

Boswell, R., Craw, S. & Rowe, R. (1997). Knowledge refinement for a design system, *in* E. Plaza & R. Benjamins (eds), *Proceedings of the Tenth European Knowledge Acquisition Workshop*, Springer, Sant Feliu de Guixols, Spain, pp. 49–64.

Brelaz, D. (1979). New methods to colour the vertices of a graph, *Communications of the ACM* **22**: 251–256.

Brodley, C. E. & Friedl, M. A. (1996). Identifying and eliminating mislabelled training instances, *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, Portland, Oregon, pp. 799–805.

Carbonara, L. & Sleeman, D. (1999). Effective and efficient knowledge base refinement, *Machine Learning* **37**: 143–181.

Cohn, D., Atlas, L. & Ladner, R. (1994). Improving generalization with active learning, *Machine Learning* **15**: 201–221.

Cohn, D., Ghahramani, Z. & Jordan, M. I. (1996). Active learning with statistical models, *Journal of Artificial Intelligence Research* **4**: 129–145.

Craw, S. (1996). Refinement complements verification and validation, *International Journal of Human-Computer Studies*, Vol. 44, pp. 245–256.

Craw, S. & Boswell, R. (1999). Representing problem-solving for knowledge refinement, *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, AAAI Press, Menlo Park, California, pp. 227–234.

Craw, S. & Hutton, P. (1995). Protein folding: Symbolic refinement competes with neural networks, *in* A. Prieditis & S. Russell (eds), *Machine Learning: Proceedings of the Twelfth International Conference*, Morgan Kaufmann, Tahoe City, CA, pp. 133–141.

Dechter, R. & Frost, D. (1999). Backtracking algorithms for constraint satisfaction problems, *Technical Report R56*, University of California Irvine.

Dechter, R. & Meiri, I. (1994). Experimental evaluation of preprocessing algorithms for constraint satisfaction problems, *Artificial Intelligence* **68**: 211–341.

Dempster, A. P., Laird, N. M. & Rubin, D. B. (1977). Maximum likelihood from incomplete data via the em algorithm, *Journal of the Royal Statistical Society, Series B* **39**: 1–38.

Dietterich, T. G. (1998). Approximate statistical tests for comparing supervised classification learning algorithms, *Neural Computation* **10**: 1895–1924.

Dietterich, T. G. (2000). Ensemble methods in machine learning, *in* J. Kittler & F. Roli (eds), *First International Work Shop on Multiple Classifier Systems, Lecture Notes in Computer Science*, Springer Verlag, New York, pp. 1–15.

Domingos, P. & Pazzani, M. (1997). On the optimality of the simple bayesian classifier under zero-one loss, *Machine Learning* **29**: 103–130.

Fisher, D. (1985). Conceptual clustering, *in* W. Klosgen & J. Zytkow (eds), *Handbook of Data Mining and Knowledge Discovery*, Oxford University Press.

Freund, Y., Seung, H. S., Shamir, E. & Tishby, N. (1997). Selective sampling using query by committee algorithm, *Machine Learning* **28**: 133–168.

Frost, D. & Dechter, R. (1994). In search of the best constraint satisfaction search, *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pp. 301–306.

Frost, D. & Dechter, R. (1995). Look-ahead value ordering for constraint satisfaction problems, *Proceedings of the Fourteenth IJCAI Conference*, pp. 572–578.

Gaschnig, J. (1979). Performance measurements and analysis of certain search algorithms, *Technical Report CMU-CS-79-124*, Carnegie-Mellon University, PA.

Gent, I., MacIntyre, E., Prosser, P., Smith, B. & Walsh, T. (1996). An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem, *in Principles and Practice of Constraint Programming*, Springer-Verlag, pp. 179–193.

Gent, I. P., MacIntyre, E. & Prosser, P. (1996). The constraindness of search, *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, Portland, Oregon.

Gil, Y. (1995). Learning from the environment by experimentation: The need for few and informative examples, *Proceedings of the AAAI Symposium on Active Learning*, MIT, Cambridge, MA.

Glover, F. & Laguna, M. (2000). Tabu search, *in* D. Corne, M. Dorigo & F. Glover (eds), *New Methods in Optimisation*, McGraw-Hill.

Hanson, S. J. (1990). Conceptual clustering and categorization, *in* Y. Kodratoff & R. S. Michalski (eds), *Machine Learning Volume III*, Morgan Kaufmann, San Mateo, CA, pp. 235–268.

Haralick, R. & Elliott, G. (1980). Increasing tree-search efficiency for constraint satisfaction problems, *Artificial Intelligence* **14**: 263–313.

Kambhampati, S. (1998). On the relations between intelligent backtracking and failure-driven explanation based learning in constraint satisfaction and planning, *Artificial Intelligence* **105**: 161–208. http://enws318.eas.asu.edu/pub/rao/jour-ddb.ps.

Kambhampati, S. (2000). Planning graph as a dynamic csp: Exploiting ebl, ddb, and other csp search techniques in graphplan, *Journal of Artificial Intelligence Research* **12**: 1–34.

Kambhampati, S. & Nigenda, R. S. (2000). Distance-based goal-ordering heuristics for graphplan, *AIPS2000*. rakaposhi.eas.asu.edu/SKambhampati00.ps.

Kodratoff, Y. (1988). *Introduction to Machine Learning*, Pitman, London.

Kondrak, G. & van Beek, P. (1997). A theoretical evaluation of selected backtracking algorithms, *Artificial Intelligence* **89**: 365–387.

Kumar, V. (1992). Algorithms for constraint satisfaction problems: A survey, *AI Magazine* **13**: 32–44.

Lamma, E., Mello, P., Milano, M., Cucchiara, R., Gavanelli, M. & Piccardi, M. (1999). Constraint propagation and value acquisition: why we should do it interactively, *Proceedings of the Fourteenth IJCAI Conference*, Sweden, Stockholm, pp. 468–473.

Langley, P., Drastal, G., Rao, R. B. & Greiner, R. (1994). Theory revision in fault hierarchies, *Proceedings of the Fifth International Workshop on Principles of Diagnosis*, New Paltz, NY.

Lewis, D. D. & Catlett, J. (1994). Heterogeneous uncertainty sampling for supervised learning, *in* W. W. Cohen & H. Hirsh (eds), *Machine Learning: Proceedings of the Eleventh International Conference*, Morgan Kauffman, San Francisco, CA, pp. 148–156.

Lindenbaum, M., Markovich, S. & Rusakov, D. (1999). Selective sampling for nearest neighbor classifiers, *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, AAAI Press, Menlo Park, California, pp. 366–371.

Maloof, M. A. & Michalski, R. D. (1995). A method for partial-memory incremental learning and its application to computer intrusion detection, *Proceedings of the 7TH IEEE International Conference on Tools with Artificial Intelligence (ICTAI'95)*, IEEE Press, Washington, DC, pp. 392–397.

Maloof, M. A. & Michalski, R. S. (1999). Selecting examples for partial memory learning, *Machine Learning* **1**: 319–342.

McCallum, A. & Nigam, K. (1998). Employing em in pool-based active learning for text classification, *Proceedings of the Fifteenth International Conference on Machine Learning*, pp. 359–367.

McCallum, A., Nigam, K. & Ungar, L. (2000). Efficient clustering of high-dimensional data sets with application to reference matching, *To appear in Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-2000).*

Mendenhall, W. & Sincich, T. (1988). *Statistics for the Engineering and Computer Sciences*, Collier MacMillan, London.

Merialdo, B. (1991). Tagging text with a probabilistic model, *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing.*

Michalski, R. S. & Chilausky, R. (1980). Learning by being told and learning from examples: An experimental comparison of the two methods of knowledge acquisition in the context of developing an expert system for soybean disease diagnosis, *International Journal of Policy Analysis and Information Systems* **4**: 125–161.

Michalski, R. S. & Stepp, R. E. (1983). Learning from observation: Conceptual clustering, *in* R. S. Michalski, J. G. Carbonell & T. M. Mitchell (eds), *Machine Learning: An Artificial Intelligence Approach*, Tioga, Palo Alto, CA, pp. 333–363.

Michalski, R. & Stepp, R. (1990). Clustering, *in* S. Shapiro (ed.), *Encyclopedia of Artificial Intelligence*, Vol. 1, Wiley, pp. 103–110.

Minitab-Inc. (1998). *MINITAB User's Guide 2: Data Analysis and Quality Tools*, Minitab Inc, State College, PA.

Minton, S., Johnston, M. D., Philips, A. B. & Laird, P. (1992). Minimising conflicts: A heuristic repair method for constraint satisfaction and scheduling problems, *Artificial Intelligence* **58**: 161–205.

Mittal, S. & Falkenhainer, B. (1990). Dynamic constraint satisfaction problems, *Proceedings of the Eighth National Conference on Artificial Intelligence*, Menlo Park, CA, pp. 25–32.

Mooney, R. J. (1992). Batch versus incremental theory refinement, *Proceedings of the AAAI Spring Symposium on Knowledge Assimilation*, Stanford, CA.

Mooney, R. J. (1997). Integrating abduction and induction in machine learning, *Appears in the Working Notes of the IJCAI-97 Workshop on Abduction and Induction in AI*, Nagoya, Japan, pp. 37–42.

Mooney, R. J. & Ourston, D. (1991). Constructive induction in theory refinement, *in* L. Birnbaum & G. Collins (eds), *Machine Learning: Proceedings of the Eighth International Workshop*, pp. 178–182.

Murphy, P. M. & Pazzani, M. J. (1994). Revision of production system rule-bases, *Machine Learning: Proceedings of the Eleventh International Conference.*

Nigam, K., McCallum, A., Thrun, S. & Mitchell, T. (1998). Learning to classify text from labeled and unlabeled documents, *AAAI98*, AAAI Press, Menlo Park, California, pp. 792–799.

Nigam, K., McCallum, A., Thrun, S. & Mitchell, T. (2000). Text classification from labeled and unlabeled documents using em, *Machine Learning* **1**. to appear.

Ourston, D. & Mooney, R. (1994). Theory refinement combining analytical and empirical methods, *Artificial Intelligence* **66**: 273–309.

Palmer, G. J. & Craw, S. (1996). The role of test cases in automated knowledge refinement, *Research and Development in Expert Systems XIII: Proceedings of Expert Systems 96, 16th Annual Technical Conference of the British Computer Society Specialist Group on Expert Systems*, SGES Publications, Cambridge, UK, pp. 75–90.

Palmer, G. J. & Craw, S. (1997). The selection of training cases for automated knowledge refinement, *in* I. J. Vanthienen & F. van Harmelen (eds), *Proceedings of the Fourth European Symposium on the Validation and Verification of Knowledge Based Systems*, Morgan Kaufmann, Leuven, Belgium, pp. 205–215.

Pazzani, M. J. & Brunk, C. A. (1991). Detecting and correcting errors in rule-based expert systems: An integration of empirical and explanation-based learning, *Knowledge Acquisition Journal* **3**: 157–173.

Pazzani, M. J. & Kibler, D. (1990). The utility of knowledge in inductive learning, *Technical Report 90-18*, University of California Irvine.

Prosser, P. (1993). Domain filtering can degrade intelligent backtracking search, *Proceedings of the Thirteenth IJCAI Conference*, Chambery, France, pp. 262–267.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*, Morgan Kaufmann, San Mateo.

Rajamoney, S. A. (1990). A computational approach to theory revision, *in* J.Shrayer & P. Langley (eds), *Computational Models of Scientific Discovery and Theory Revision*, Morgan Kaufmann, San Francisco, pp. 225–257.

Rasumssen, E. (1992). Clustering algorithms, *in* W. B. Frakes & R. Baeza-Yates (eds), *Information Retrieval: Data Structures and Algorithms*, Prentice Hall, London, pp. 419–442.

Richards, B. & Mooney, R. (1991). First-order theory revision, *in* L.A.Birnbaum & G.C.Collins (eds), *Machine Learning: Proceedings of the Eighth International Workshop*, Morgan Kauffman, San Mateo, CA, pp. 447–451.

Richards, B. & Mooney, R. (1995). Automated refinement of first-order horn-clause domain theories, *Machine Learning* **19**: 95–131.

Sadeh, N. M. & Fox, M. S. (1990). Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem, *Proceedings of the Fourth International Conference on Expert Systems in Production and Operations Management*, pp. 134–144.

Schlimmer, D. F. . J. (1988). Concept simplification and predictive accuracy, *Machine Learning: Proceedings of the Fifth International Conference*, Morgan Kaufmann, pp. 22–28.

Schlimmer, J. C. & Fisher, D. (1986). A case study of incremental concept induction, *Proceedings of the Fourth National Conference on Artificial Intelligence*, Philadelphia, PA, pp. 496–501.

Scott, P. D. & Markovitch, S. (1989). Uncertainty based selection of learning experiences, *Proceedings of The Sixth International Workshop on Machine Learning*, Morgan Kaufmann, Ithaca, New York.

Selman, B. & Kautz, H. (1993). Domain independent extensions to GSAT: Solving large structured satisfiability problems, *Proceedings of the Thirteenth IJCAI Conference*, Chambery, France.

Shannon, C. E. & Weaver, W. (1949). *The Mathematical Theory of Communication*, University of Illinois Press.

Smith, B. & Grant, S. (1998). Trying harder to fail first, *Proceedings of the ECAI98 Conference*, John Wiley and Sons Ltd., Brighton, UK, pp. 249–253.

Tallis, M. & Gil, Y. (1999). Designing scripts to guide users in modifying knowledge based systems, *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, AAAI Press, Menlo Park, California, pp. 227–234.

Tsang, E. (1993). *Foundations of Constraint Satisfaction*, Academic Press, San Diego.

Tsang, E., Borrett, J. & Kwan, A. (1994). An attempt to map the performance of a range of algorithms and heuristic combinations, *Technical Report AUCS/TRCSM-210*, University of Essex. ftp.essex.ac.uk/pub/csc/technical-reports/CSM-210.ps.

van Rijsbergen, C. J. (1980). *Information Retrieval*, Butterworths, London.

Wilkins, D. C. (1988). Knowledge base refinement using apprenticeship learning techniques, *Proceedings of the Sixth National Conference on Artificial Intelligence*, Minnesota, pp. 646–651.

Wilkins, D. C. (1990). Knowledge base refinement as improving an incorrect and incomplete domain theory, *in* Y. Kodratoff & R. S. Michalski (eds), *Machine Learning Volume III*, Morgan Kaufmann, San Mateo, CA, pp. 493–513.

Willett, P. (1988). Recent trends in hierarchic document clustering: A critical review, *Information Processing and Management* **24**: 577–597.

Wiratunga, N. & Craw, S. (1999a). Incorporating backtracking search with knowledge refinement, *in* A. Vermesan & F. Coenen (eds), *Proceedings of the Sixth European Symposium on the Validation and Verification of Knowledge Based Systems*, Kluwer Academic Publishers, Oslo, Norway, pp. 193–205.

Wiratunga, N. & Craw, S. (1999b). Sequencing training examples for iterative knowledge refinement, *Proceedings of the Nineteenth SGES International Conference on Knowledge Based Systems and Applied Artificial Intelligence*, Springer, Cambridge, UK, pp. 41–56.

Wiratunga, N. & Craw, S. (2000). Informed selection of training examples for knowledge refinement, *in* R. Dieng & O. Corby (eds), *Proceedings of the Twelfth International*

Conference on Knowledge Engineering and Knowledge Management, Springer, Juan-les-Pins, France, pp. 233–248.

Wogulis, J. & Pazzani, M. J. (1993). A methodology for evaluating theory revision systems: Results with audrey ii, *Journal of Artificial Intelligence Research* .

Zlatareva, N. & Preece, A. (1994). State of the art in automated validation of knowledge-based systems, *Expert Systems with Applications* **7**: 151–167.

Appears in Proceedings of the 5th European Symposium on Verification and Validation of Knowledge Based Systems and Components (EUROVAV99), pp 193-205,

INCORPORATING BACKTRACKING IN KNOWLEDGE REFINEMENT

Nirmalie Wiratunga and Susan Craw
School of Computer and Mathematical Sciences
The Robert Gordon University
Aberdeen AB25 1HG, UK
nw|smc@scms.rgu.ac.uk

Abstract Refinement tools seek to correct faulty rule-based systems by identifying and repairing faults that are indicated by training examples that provide some evidence of faults. Refinement tools typically use a hill-climbing search to identify suitable repairs. In this paper, the goal is to incorporate an effective backtracking mechanism with a refinement algorithm so that the search for repairs does not get caught by local maxima. However the repair cycle for each potential fault is expensive, so exhaustive backtracking is prohibitive for large knowledge bases. This paper investigates more guided backtracking algorithms developed for constraint satisfaction problems and adapts them for refinement problems. Experiments with these backtracking algorithms reveal that high accuracy refined knowledge bases are achievable, often at the expense of extra iterations, but an informed re-ordering of training examples reduces the number of iterations without increasing the error-rate. A test-bed is developed by corrupting a rule base with interacting faults, thereby allowing pairs of conflicting training examples to be identified. The algorithms are evaluated on training sets containing increasing numbers of these conflicting examples. One separate observation is that conflicting examples help to achieve refined knowledge bases with high accuracy.

Keywords: Knowledge Refinement, Informed Backtracking, Example Re-ordering

1. INTRODUCTION

Refinement tools support the knowledge acquisition and development of knowledge based systems (KBSs) by assisting the debugging of incorrect systems and the adaptive maintenance of KBSs whose problem-

solving environment changes (Craw, 1996; Boswell et al., 1997). Refinement tools are commonly presented with examples of problem-solving where the expert's solution is inconsistent with the KBS's, and from these, the tool identifies potential faults in the KBS and suggests possible repairs. It also benefits from knowing some correctly solved examples as well, so that repairs are not too closely fitted to wrongly-solved examples only, to the detriment of the KBS's more general problem solving. Therefore the training set for the refinement tool's learning contains a selection of wrongly and correctly solved examples, each consisting of the facts that describe the problem-solving task, together with the expert's solution for this task.

Refinement tools adopt an incremental approach where each application of the algorithm attempts to fix one or more, but typically not all, of the wrongly-solved examples in the training set, and to improve the accuracy on the training set with a view to improving the accuracy more generally. The refinement task is sufficiently complex that the space of possible repairs demands a heuristic search, typically hill-climbing. EITHER (Ourston and Mooney, 1994) and FORTE (Richards and Mooney, 1995) try to repair the outstanding fault that is indicated by the *largest* number of examples, and choose the repair with the *fewest* changes to rules which are *nearest* the observables. KRUSTTools are KBS specific refinement tools, assembled from our KRUSTWorks generic refinement toolkit. The refinement algorithm central to this family also applies a hill-climbing search. Although it generates many refined KBSs designed to fix each incorrect example, it then chooses the refined KBS with the *highest* accuracy on the training examples as the input KBS for the next iteration of the algorithm. The result is that refinement tools are dogged by the standard hill-climbing problem of getting caught in local maxima, so the accuracy or performance of the KBS must be reduced before an overall improvement can be gained.

In this paper we explore different ways KRUSTTools may exploit previously abandoned repairs or refined KBSs to restart the refinement process when it gets stuck. First, we illustrate situations when KRUSTTools fail to generate refined KBSs and indicate how backtracking is applied. More selective backtracking algorithms, developed to solve constraint satisfaction problems (CSPs), are presented next, and these are then adapted to fit the KRUSTTool refinement cycle. Experimental results suggest the need for refinement-specific improvements to the basic backjumping algorithms and these changes are presented and evaluated. Finally we conclude with a few general observations and directions for future work.

2. REFINEMENT WITH KRUSTTOOLS

A KRUSTTool incrementally refines a KBS by processing the training examples $\{e_1, \ldots, e_n\}$ one at a time, Figure 1. The input KBS is the best refined KBS from the previous iteration, or the original faulty KBS for the first iteration. In each iteration the next training example, called the *refinement example* for this iteration, is used to generate refined KBSs. If the expert's solution for the refinement example already coincides with the input KBS's solution then no refinement is necessary in this cycle. Otherwise, the refinement example's evidence allocates blame to possible faults in the KBS and generates potential repairs that are implemented as the refined KBSs proposed during this cycle. Two data structures of examples provide a selection mechanism for the best refined KBS. The *constraint examples buffer* contains the previous refinement examples that have already been corrected, and refined KBSs are rejected if they wrongly answer any constraint example in this buffer. The *training examples buffer* contains the training examples still to be processed, and the remaining refined KBSs are ranked by their accuracy on the training examples buffer; the previous filter guarantees 100% accuracy on the constraint examples buffer. During each cycle, the current refinement example is transferred from the training examples buffer into the constraint examples buffer.

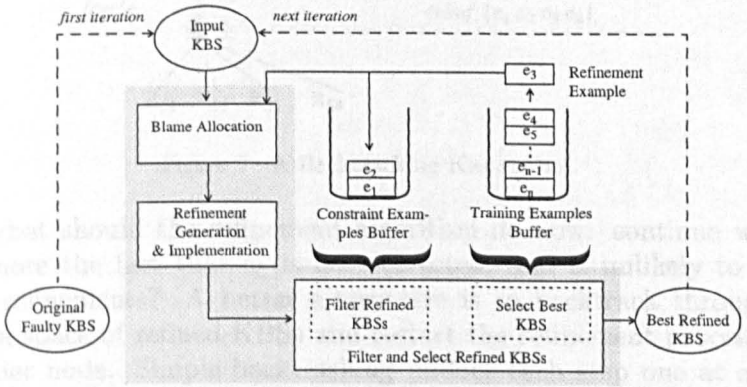


Figure 1 The KRUSTTool Process.

Our refinement algorithm is unusual in generating many refined KBSs in each iteration, and the hill-climbing selection of the *one* best refined KBS for the next iteration occurs at the *end* of each cycle. This offers the possibility of backtracking to alternative refined KBSs thereby achieving a best-first search. Figure 2 illustrates the start of a potential backtracking scenario; the updates to the constraint examples buffer

(cebuf) and the training examples buffer (tebuf) are shown on the right. Refinement example e_2 generates 3 refined KBSs and R_{21} is selected as best. Refinement examples e_3 and e_4 generate several refined KBSs and again the best is selected. But now suppose R_{41} cannot be refined by e_5 because although 4 refinements are generated, all are rejected by the constraint examples; this is shown by a darkly shaded node for e_5. The *refinement path* in the diagram is ... $e_2.R_{21} \rightarrow e_3.R_{31} \rightarrow e_4.R_{41} \rightarrow e_5.\emptyset$ where $\emptyset$ indicates the absence of a selected refined KBS. Strictly, it is this refinement path that labels the nodes in the diagram and so the node labelled R_{51} is really named $R_{...21314151}$.

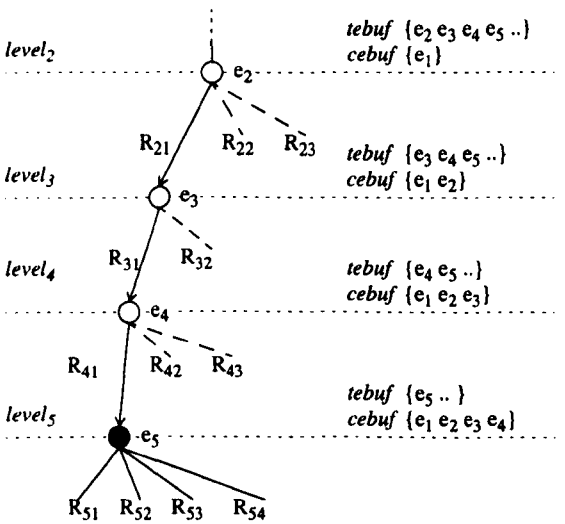


Figure 2 A Backtracking KRUSTTool.

So what should the refinement algorithm do now: continue with e_6 and ignore the fact that e_5 is not corrected, and is unlikely to be by future refinements? A better alternative is to backtrack through the solution space of refined KBSs and restart the refinement process from an earlier node. Simple backtracking undoes each step one at a time, and so refinement is restarted with R_{42} and e_5. In this paper we investigate more guided backtracking that may restart refinement from earlier points, say R_{22} with e_3. These algorithms originated as search methods for solving constraint satisfaction problems and are introduced next.

3. HEURISTIC SEARCH SOLVES CSP

Constraint satisfaction problems (CSPs) consist of a set of ordered variables $\{v_1, \ldots, v_n\}$, a specified domain D_i for each variable v_i and a

set of constraints $\{C_1, \ldots, C_m\}$. A CSP solution is an instantiation of each variable with a value from its respective domain such that none of the constraints is violated (Tsang, 1993). Various backtracking searches have been proposed that partially undo the instantiation and resume the constructive process from a previous variable instantiation.

BackTracking (BT) (Bitner and Reingold, 1975) steps back to the previous variable v_{i-1}, and continues the search by finding a new instantiation for v_{i-1} consistent with the constraints and v_k, $k < i-1$. BT recursively backtracks to previous variables until it has tried all values in the domain for each.

BackJumping (BJ) (Gaschnig, 1979) does not step back to the previous variable v_{i-1} but instead jumps back to the latest variable v_j whose instantiation conflicts with any of the instantiations for v_i. If there are no new instantiations available for v_j then BJ reverts to backtracking from v_j.

Conflict-directed BackJumping (CBJ) (Prosser, 1993) extends the notion of backjumping by replacing the backtracking after a backjump in BJ with backjumping.

BT is an exhaustive depth first search of the tree of variable instantiations; siblings are different instantiations of a particular variable and a parent instantiates the preceding variable in the given ordering. BJ explores a subset of the BT nodes and so our motivation for investigating backjumping is to reduce the number of refinement iterations.

BJ and CBJ are no longer exhaustive. However, for binary CSPs, where all constraints contain at most 2 variables, BJ and CBJ still find all solutions (Kondrak and van Beek, 1997); any instantiations they fail to check for variables between v_i and the backjumped to v_j are guaranteed to result in the same inconsistency between the instantiation for v_j and the possible values for v_i. Therefore, for binary CSPs, BJ and CBJ have proved effective in reducing search.

4. CSP ALGORITHMS AID REFINEMENT

We wish to adapt the CSP algorithms to search the space of incrementally refined KBSs created by KRUSTTools, so that the KRUSTTool, when necessary, may revisit refined KBSs that have previously been abandoned by the refinement algorithm. We propose an analogy between CSPs and refinement problems so that the concepts applied in the CSP algorithms can be imitated in the refinement domain.

In refinement problems we incrementally refine the KBS to correctly answer the current and previous refinement examples. So, the most natural analogy between CSPs and refinement links variables with training examples, the current variable with the refinement example, and instantiated variables with correctly solved training examples in the constraint examples buffer. CSP constraints correspond to refined KBSs, and consistency is achieved when the refined KBS correctly answers the constraint examples. Finally the domain for a variable corresponds to the repairs that are proposed by a refinement example.

To complete the analogy we must describe when backtracking is triggered and how backjumps are determined. The KRUSTTool algorithm fails when the refinement example e_i and the input KBS R fail to create any refined KBSs (i.e. the generated KBSs $Generated_{Ri}$ is empty) or those generated are rejected by the constraint examples (i.e. $Filtered_{Ri}$ is empty). The *conflict set* for e_i, $confset(e_i)$, will contain the potential backtracking points from e_i. If $Filtered_{Ri} = \{\}$ then we know which constraint examples caused the removal of each generated KBS, and these form the confset for the CBJ algorithm. BJ's confset is similar but also contains refinement examples prior to the conflicting ones. If $Generated_{Ri} = \{\}$ then backtracking is the only option; no conflicting constraint examples can be identified since there are no KBSs to test!

Let us revisit Figure 2's scenario. Refinement must backtrack because $Filtered_{R415} = \{\}$, although $Generated_{R415} = \{R_{51}, R_{52}, R_{53}, R_{54}\}$. Thus for each KBS in $Generated_{R415}$, some of the constraint examples in cebuf must be wrongly answered; suppose R_{51}, R_{52} wrongly solve e_2, and R_{53}, R_{54} wrongly solve e_3. For BT, e_5's conflict set is the previous refinement example $\{e_4\}$ and refinement proceeds by backtracking to e_4 on the refinement path and choosing the next branch; in this case R_{42} with e_5. For BJ and CBJ, e_5's conflict set contains the failed constraint examples e_2, e_3. So refinement continues from e_3, the most recent on the path, selecting the next available refined KBS R_{32} with e_4 as the refinement example; e_5 is moved back into tebuf as a future refinement example. If no more KBSs are available from e_3 then BJ backtracks to the e_2 node and CBJ backjumps according to e_3's and e_5's conflict sets.

4.1 REFINEMENT DIFFERS FROM CSP

We have drawn an analogy between CSPs and knowledge refinement that allows us to apply backtracking and backjumping algorithms with the KRUSTTool algorithm. However, there are two obvious differences between CSPs and refinement: the domain of potential repairs is not known in advance, instead it is constructed incrementally during refine-

ment generation and filtering; and the behaviour of constraint examples can change – they can become uncorrected and so they provide new fault evidence. The first is dealt with by associating refined KBSs with the refinement examples that generate them, and reasoning about backtracking using constraint examples rather than KBSs.

Figure 3 illustrates a problem that can arise from the second point. In this scenario, R_{21} already answers e_3 correctly and so the output from the e_3 cycle is the input KBS R_{21}; this has been highlighted by light shading. It does not affect the search when it is advancing, but backtracking or backjumping to this point raises problems. In Figure 3, backtracking starts because $Filtered_{R_{41}5} = \{\}$. Suppose we are using BJ and $confset(e_5)$ is $\{e_2, e_3\}$, so we backjump to e_3. But the input KBS R_{21} already correctly answers e_3 and so no refined KBSs are available. We could simply backtrack further, but the refinement tool has just discovered a relationship: the changes to correct e_5 have interacted with the way that e_3 was previously proved. Thus if we backtrack beyond e_3 then it is possible that the same interaction will occur again.

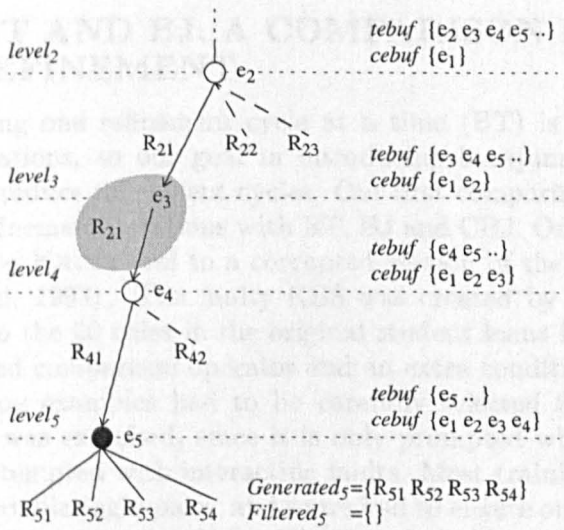


Figure 3 The Changing Behaviour of Constraint Examples.

Instead, we note these special examples and treat them differently. We call e_3 a *latent* example since it did not contribute any fault evidence as a refinement example. The other refinement examples are active. Given the conflicting relationship between the latent example e_3 and its activating refinement example e_5, we choose to solve their conflict at this

point by re-instating e_3 in to tebuf and advancing the search from R_{51} with e_3 as the next refinement example.

4.2 PRIORITISING LATENT OVER ACTIVE

Latent examples provide no impact on the refinement initially since it is already answered correctly. But when it appears in the conflict set, not only does it provide fault evidence, but it has the added conflicting relationship with the current refinement example. We have amended the backjumping algorithms to take further account of latent examples in conflict sets. If in Figure 3, $confset(e_5)$ is $\{e_3, e_4\}$, then backjumping will resume with e_4 and the fault evidence now presented by the latent example $e3$ will be lost. Instead, we prioritise latent examples that appear in conflict sets, and, rather than backjumping to the most recent conflicting example, we reinstate all conflicting latent examples into the tebuf. In Figure 3 the search proceeds with e_3 and R_{51}, the refined KBS in $Generated_{R_{41}5}$ with the highest accuracy, despite e_4 being in the conflict set. If the intervening active conflict examples (here e_4) remain a problem, backjumping offers the opportunity to investigate there later.

4.3 BT AND BJ: A COMPARISON FOR REFINEMENT

Backtracking one refinement cycle at a time (BT) is likely to lead to many iterations, so our goal in introducing backjumping (BJ and CBJ) was to reduce refinement cycles. Our first comparison counts the number of refinement iterations with BT, BJ and CBJ. Our experiments apply a Prolog KRUSTTool to a corrupted version of the student loans KBS (Pazzani, 1993). The faulty KBS was created by introducing 5 corruptions to the 20 rules in the original student loans KBS: an extra rule, a changed comparison operator and an extra condition in 3 rules.

The training examples had to be carefully selected to ensure that backtracking was exercised, since it is only prompted when conflicting repairs are attempted with interacting faults. Most training sets do not require such conflicting repairs, and so we had to ensure our training sets did indeed contain some conflicts. We identified 9 conflicting pairs in a carefully chosen set of 8 examples from the complete student dataset, where repairs for one example in the pair conflicted with repairs for the other. Finding conflicting examples was relatively easy given the density of corruption of the KBS. Our selected dataset contained a further 22 "normal", unconflicting examples. Training sets of a given conflict level N were created from the selected dataset of 30 examples by randomly choosing N conflict pairs, removing duplicate examples when they oc-

curred, and randomly selecting from the "normal" examples until the training set contained 15 examples. KRUSTTools incorporating the BT, BJ and CBJ algorithms were applied to each training set and the corrupted KBS. Each test was repeated 10 times and the results averaged.

Figure 4 shows the number of iterations for each of the algorithms as the number of conflict pairs in the training set increases. The results were surprising. We had expected BT to have the most iterations, BJ to have fewer, and CBJ to have the fewest, reflecting the increased targeting of the search. With binary CSPs, BT is guaranteed to have at least as many iterations as BJ or CBJ. However, in the more dynamic space of refined KBSs this is not the case; backjumping searched a different part of the space that involved more iterations.

So has there been any gain from BJ's additional searching? Figure 5 shows the error rates of the final KBS produced by the 3 algorithms on the complete set of 30 examples; the error-rate of the original corrupted KBS is the horizontal dashed line on all error-rate graphs. BJ, the most greedy in refinement cycles, has indeed gained the lowest error rate. This behaviour is explained by noticing that, although BJ and CBJ are guaranteed to find all binary CSP solutions, this is not the case with refinement, since repairs in different cycles can interact: an earlier repair can provide part of a later repair or conflict with the later repair. Therefore the repairs that are proposed depend on the input KBS and thus the refinement path.

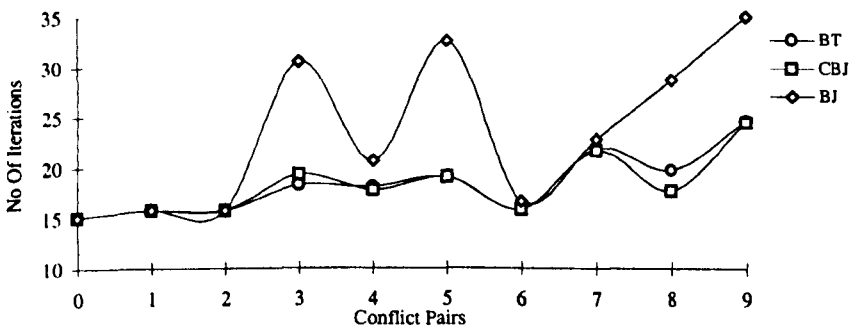


Figure 4 Number of Iterations (Basic Algorithms).

5. CONFLICT-BASED RE-ORDERING

Figure 5 shows another interesting trend: the error rate of the refined KBS decreases as the number of conflict pairs in the training set increases. This confirms the experimental results in (Palmer and Craw,

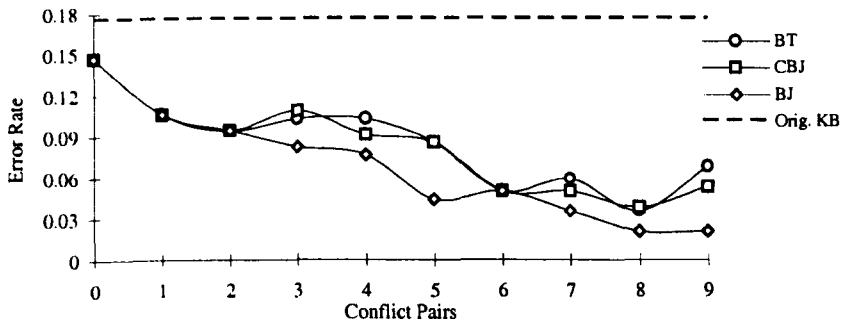


Figure 5 Error Rate of Final Refined KBS (Basic Algorithms).

1996), that the more demanding the examples in the training set the higher accuracy achieved by refinement. It also suggested that we explore re-ordering the training examples to exploit conflict knowledge as soon as it is recognised. Minimal Bandwidth Ordering heuristic for static ordering of variables attempts to reduce the backtracking distance for CSP algorithms by placing mutually constrained variables close together in the search (Tsang, 1993). The previous section recognised that the refinement example and the conflicting examples are mutually constraining since the repairs for the later one has affected the correctness of the earlier latent example. We try to use this idea of mutually constraining examples to associate the refinement example and the deepest conflicting constraint example in the sequence of training examples in an attempt to reduce the number of iterations of the backjumping algorithms without compromising the error-rate of the final refined KBS.

Figure 6 illustrates a hypothetical backjumping situation. The refinement search space contains three main refinement paths, of which two have been discarded: $e_2.R_{21} \rightarrow e_3.R_{31} \rightarrow e_4.R_{41} \rightarrow e_5.\emptyset$ and $e_2.R_{22} \rightarrow e_3.R_{31} \rightarrow e_4.R_{41} \rightarrow e_5.\emptyset$. Suppose in each case $confset(e_5)= \{e_2\}$ and so backjumping to e_2 produces the search as illustrated. But this also means that e_2 and e_5 are mutually constraining since the repairs to e_5 has affected the solution to e_2.

The Minimal-BJ (MBJ) and Minimal-CBJ (MCBJ) algorithms contain a further amendment to the backjumping algorithms, so that backjumping to a node e_j that conflicts with the current refinement example e_i causes the algorithm to try to fix this pair of mutually constraining examples next. It re-sorts tebuf so that e_i is re-used immediately with the next refined KBS from e_j. Thus the pair of conflicting examples identified in backjumping become adjacent on the new branch of the

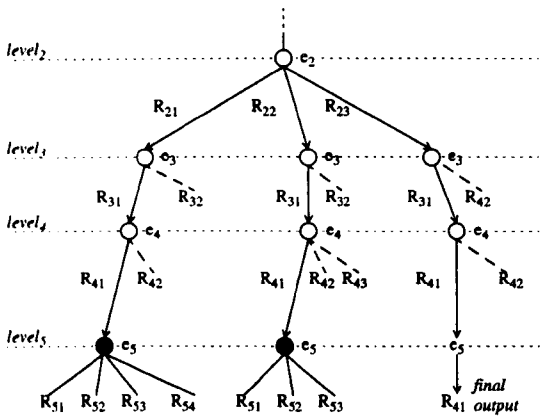


Figure 6 Searching without Conflict-Based Re-Ordering.

refinement path. Figure 7 illustrates a possible outcome of re-ordering the tebuf examples so that e_5 is used as the next refinement example after backjumping to e_2, and indicates the potential saving in iterations over Figure 6.

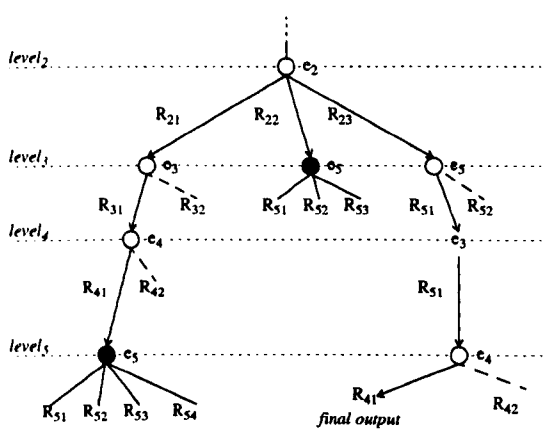


Figure 7 Searching with Conflict-Based Re-Ordering.

Although this re-ordering is not guaranteed to reduce iterations, the relationship between an example and its conflict set gives some justification for re-ordering the otherwise random order of the training examples. It is possible that successive re-ordering of nodes in this manner may at times lead to the original sequence. Even so, this will not result in cycling because BJ and CBJ will resort to backtracking once all branches of a node are explored.

We included the MBJ and MCBJ algorithms in our earlier experiments. Figure 8 superimposes the barchart for MBJ iterations on the line graphs for the basic algorithms; the results for MCBJ are similar to CBJ's so are not shown on the graph. Our goal of reducing the number of iterations in BJ has been achieved in general and MBJ's iterations are closer to BT and CBJ. There were 3 test runs where BJ performed fewer iterations than MBJ, and a closer examination of one indicated that re-ordering resulted in an increased search space when two examples e_i and e_j are affected by the same repair, where the fault evidence provided by e_j cannot be fixed before the fault evidence from e_i is fixed. Dependencies of this nature suggest the existence of a new type of constraint, and we intend to investigate ways to identify these in the future.

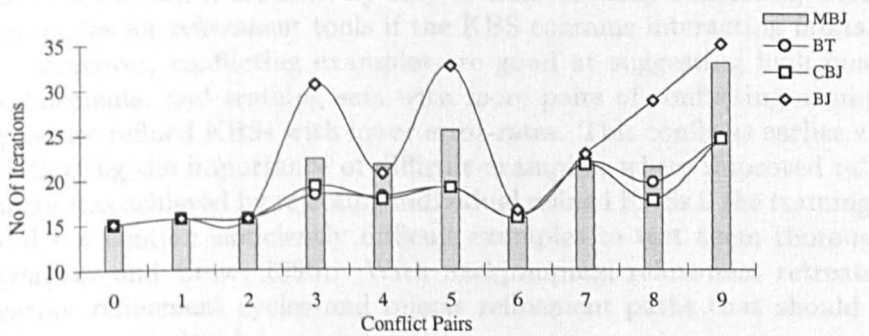


Figure 8 Number of Iterations (Conflict-Based Re-Ordering).

Figure 9 confirms that the refined KBS error rates with MBJ, and CMBJ, are unaffected by the dynamic re-ordering. So MBJ has achieved fewer iterations without increasing the error-rate of the final KBS.

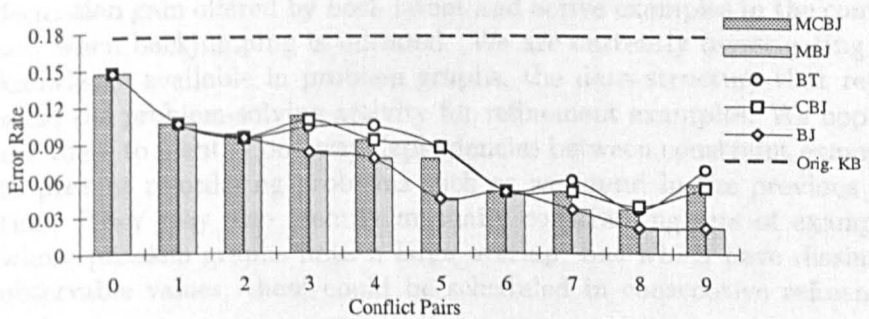


Figure 9 Error Rate of Final Refined KBS (Conflict-Based Re-Ordering).

6. CONCLUSIONS

We have transformed the natural hill-climbing of the KRUSTTool refinement algorithm into a best first search that reconsiders previously filtered out refined KBSs. It is the KRUSTTool's ability to generate many potential refined KBSs in response to fault evidence that enables CSP search strategies to be applied with the central refinement algorithm. The authors of other refinement algorithms (Ourston and Mooney, 1994; Richards and Mooney, 1995) have argued that the choice of repairs available to their tool is sufficiently flexible that hill-climbing problems occur rarely, and so make no attempt to deal with it. However dealing with mutually conflicting examples in a single refinement iteration is difficult, and otherwise hill-climbing problems arise. Our testbed has shown that it is relatively easy to find mutually conflicting training examples for refinement tools if the KBS contains interacting faults.

Moreover, conflicting examples are good at suggesting high quality refinements, and training sets with more pairs of conflicting examples produce refined KBSs with lower error-rates. This confirms earlier work indicating the importance of difficult examples, where improved refinement was achieved by rejecting individual refined KBSs if the training set did not contain sufficiently difficult examples to test them thoroughly (Palmer and Craw, 1996). With backjumping, refinement retreats to earlier refinement cycles and rejects refinement paths that should not have been explored.

Introducing backjumping to reduce the search in standard chronological backtracking reveals an interesting refinement phenomenon. The more selective backjumping may actually increase the search. However, we discovered the extra iterations are used profitably and provide a refined KBS with a lower error-rate. Amendments to the backjumping algorithms to reduce the iterations, whilst maintaining the low error-rate, concentrate on re-ordering the training examples by recognising the information gain offered by both latent and active examples in the conflict set, when backjumping is initiated. We are currently investigating the knowledge available in problem graphs, the data structure that represents the problem-solving activity for refinement examples. We hope to use these to identify one-way dependencies between constraint examples to prevent re-ordering problems such as we found in the previous section. They may also identify mutually constraining sets of examples, whose problem graphs have a large overlap, but which have dissimilar observable values; these could be scheduled in consecutive refinement cycles.

This work highlights the variety of refinement paths and re-ordering mechanisms open to refinement tools and has drawn our attention to relationships between training examples that may allow us to direct the refinement process towards staged goals in the identification and repair of KBS faults.

Acknowledgments

The KRUSTWorks project is supported by EPSRC grant GR/L38387 awarded to Susan Craw and ORS grant 98131005 awarded to Nirmalie Wiratunga. We also thank IntelliCorp Ltd for its contribution to this project.

References

Bitner, J. R. and Reingold, E. (1975). Backtrack programming techniques. *Communications of the ACM*, 18:651–656.

Blake, C., Keogh, E., and Merz, C. (1998). UCI repository of machine learning databases. www.ics.uci.edu/~mlearn/MLRepository.html.

Boswell, R., Craw, S., and Rowe, R. (1997). Knowledge refinement for a design system. *Proceedings of the Tenth European Knowledge Acquisition Workshop*, pages 49–64, Sant Feliu de Guixols, Spain. Springer.

Craw, S. (1996). Refinement complements verification and validation. In *Int. Journal of Human-Computer Studies*, 44:245–256.

Gaschnig, J. (1979). Performance measurements and analysis of certain search algorithms. Technical Report CMU-CS-79-124, Carnegie-Mellon University, PA.

Kondrak, G. and van Beek, P. (1997). A theoretical evaluation of selected backtracking algorithms. *Artificial Intelligence*, 89:365–387.

Ourston, D. and Mooney, R. (1994). Theory refinement combining analytical and empirical methods. *Artificial Intelligence*, 66:273–309.

Palmer, G. J. and Craw, S. (1996). The role of test cases in automated knowledge refinement. In *Proceedings ESs 96, Annual Conference of the BCSG on ESs*, pages 75–90, Cambridge, UK. SGES Publications.

Pazzani, M. J. (1993). Student loan relational domain. In UCI Repository of Machine Learning Databases (Blake et al., 1998).

Prosser, P. (1993). Domain filtering can degrade intelligent backtracking search. In *Proceedings of the Thirteenth IJCAI Conference*, pages 262–267, Chambery, France.

Richards, B. and Mooney, R. (1995). Automated refinement of first-order horn-clause domain theories. *Machine Learning*, 19:95–131.

Tsang, E. (1993). *Foundations of Constraint Satisfaction.* Academic Press, San Diego.

Appears in Proceedings of the 19th SGES International Conference on Knowledge Based Systems and Applied Artificial Intelligence,

Sequencing Training Examples for Iterative Knowledge Refinement

Nirmalie Wiratunga and Susan Craw
School of Computer and Mathematical Sciences
The Robert Gordon University,
St Andrew Street, Aberdeen AB25 1HG, Scotland, UK
Email: nw|smc@scms.rgu.ac.uk

Abstract

Refinement tools seek to correct faulty knowledge based systems (KBSs) by identifying and repairing faults that are indicated by training examples for which the KBS gives an incorrect solution. Refinement tools typically use a hill-climbing search to identify suitable repairs. Backtracking search algorithms, developed for constraint satisfaction problems, have been incorporated with an iterative knowledge refinement tool, to solve local maxima problems. This paper investigates how the efficiency of such a tool can be improved and introduces new and general heuristics for ordering training examples. Experimental results reveal that these heuristics applied to static and dynamic ordering of training examples can significantly improve the efficiency of the iterative refinement tool, without increasing the error-rate of the final refined KBS.

1 Introduction

Refinement tools support the knowledge acquisition and development of knowledge based systems (KBSs) by assisting the debugging of incorrect systems and the adaptive maintenance of KBSs whose problem-solving environment changes [3]. Refinement tools are presented with examples that indicate there are one or more faults in the KBS; these are often examples of problem-solving where the expert's solution is inconsistent with the KBS's solution. The tool also benefits from knowing some correctly solved examples as well, so that repairs are not too closely fitted to wrongly-solved examples only, to the detriment of the KBS's more general problem solving. Therefore the training set for the refinement tool's learning contains a selection of wrongly and correctly solved examples, each consisting of the facts that describe the problem-solving task, together with the expert's solution.

Refinement tools adopt an incremental approach where each cycle attempts to fix one or more, but typically not all, of the wrongly-solved examples in

the training set, and to reduce the error-rate on the training set with a view to reducing the error-rate more generally. The refinement task is sufficiently complex that the space of possible repairs demands a heuristic search, typically hill-climbing. EITHER [12] and FORTE [14] try to repair the outstanding fault that is indicated by the *largest* number of examples, and choose the repair with the *fewest* changes to rules which are *nearest* the observables. KRUSTTools are KBS specific refinement tools, assembled from our KRUSTWorks generic refinement toolkit and the refinement algorithm central to this family also applies a hill-climbing search. Although it generates many refined KBSs designed to fix each incorrect example, it then chooses the refined KBS with the *lowest* error-rate on the training examples as the input KBS for the next iteration. The result is that refinement tools are dogged by the standard hill-climbing problem of getting caught in local maxima, so the performance of the KBS must be reduced before an overall improvement can be gained.

In previous work we described how informed backtracking search algorithms from Constraint Satisfaction Problems (CSPs) can be incorporated within knowledge refinement so that KRUSTTools may exploit previously abandoned repairs when the refinement process comes to a halt [18]. In this paper we investigate how the efficiency of such search algorithms can be improved. Section 2 illustrates situations when KRUSTTools fail to generate refined KBSs and indicates how backtracking search is applied. We introduce concepts from CSPs and outline the search algorithm that proved best for knowledge refinement in Section 3. Heuristics to improve efficiency are discussed in Section 4 and experimental results are presented in Section 5. We conclude with directions for future work in Section 6.

2 Refinement with a KRUSTTool

A faulty KBS is incrementally refined by a KRUSTTool based on the fault evidence provided by examples $e_1, \ldots, e_n$ (Figure 1). This process is iterative with examples utilized one at a time. The input KBS for each iteration is the best refined KBS from the previous iteration, or the original faulty KBS in the first iteration. The *training examples buffer* contains all examples that are yet to be used by the KRUSTTool, and the top most example in this buffer at each iteration is chosen as the *refinement example* and drives that refinement cycle. If the refinement example is correctly solved then refinement is not required, otherwise the fault evidence is employed to allocate blame, generate refinements and implement them as refined KBSs. The refinement example is then transfered into the *constraint examples buffer*, containing all previously solved examples. The constraint examples buffer helps filter the potential refined KBSs, by rejecting those that incorrectly answer any of the constraint examples. The filtered refined KBSs are then ranked by their error-rate on the training examples buffer. Consequently, the refined KBS with the lowest error-rate is the best refined KBS for this iteration.

The KRUSTTool algorithm is unusual in generating many refined KBSs in

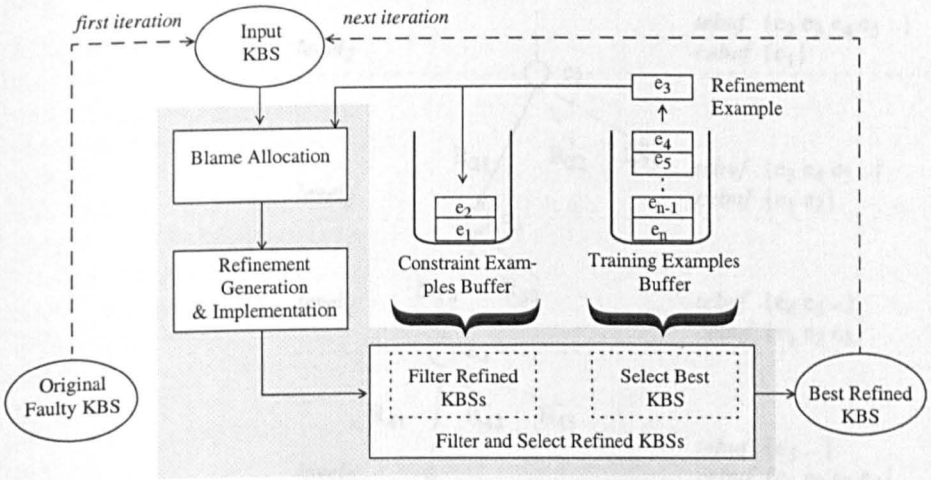


Figure 1: The KRUSTTool Process.

each iteration, and the hill-climbing selection of the *one* best refined KBS for the next iteration occurs at the *end* of each cycle. This offers the possibility of backtracking to alternative refined KBSs thereby achieving a best-first search. Figure 2 illustrates the start of a potential backtracking scenario; the updates to the training examples buffer (tebuf) and the constraint examples buffer (cebuf) are shown on the right. Refinement example e_2 generates 3 refined KBSs and R_{21} is selected as best. Refinement examples e_3 and e_4 generate several refined KBSs and again the best is selected. But now suppose R_{41} cannot be refined by e_5 because although 4 refinements are generated, all are rejected by cebuf; this is shown by a darkly shaded node for e_5.

So what should the refinement algorithm do now: continue with e_6 and ignore the fact that e_5 is not corrected, and is unlikely to be by future refinements? A better alternative backtracks through the solution space of refined KBSs and restarts the refinement process from an earlier node. Simple backtracking undoes each step one at a time, and so refinement is restarted with R_{42} and e_5. In the next section we look at a more informed backtracking that helps restart refinement from earlier points when appropriate, say R_{22} with e_3.

3 Informed Backtracking

We borrow an approach developed to direct the backtracking search for solutions to constraint satisfaction problems (CSPs). We outline the CSP method and then draw an analogy between CSPs and knowledge refinement so that we can adapt the method for knowledge refinement.

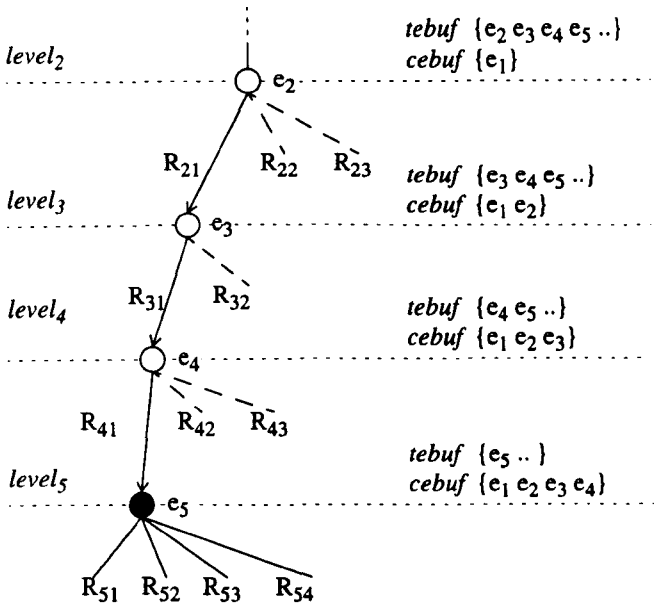


Figure 2: A Backtracking KRUSTTool.

3.1 Heuristic Searches Solve CSPs

CSPs consist of a set of ordered variables $\{v_1, \ldots, v_n\}$, a specified domain for each variable v_i and a set of constraints. A constraint is a relation defined on a subset of variables, specifying all simultaneous value assignments within this subset that are forbidden by this constraint. A CSP solution is an instantiation of each variable with a value from its respective domain such that none of the constraints is violated [17]. CSPs are often solved by constructive algorithms where each variable is instantiated in turn, so that the constraints are satisfied for this and the previous variable instantiations.

However, this hill-climbing approach may also fail: the next variable v_i may not be instantiated without violating the constraints involving v_i and the previous variables $v_1, \ldots, v_{i-1}$. Various backtracking searches have been proposed that partially undo the instantiation and resume the constructive process from a previous variable instantiation. We shall re-use chronological backtracking (BT) [1] and backjumping (BJ) [7]. Unlike BT, BJ does not step back to the previous variable v_{i-1} but instead jumps back to the latest variable v_j whose instantiation conflicts with any of the instantiations for v_i. If there are no new instantiations available for v_j then BJ reverts to backtracking from v_j. With binary CSPs, where all constraints contain at most 2 variables BJ will still find all solutions [10].

3.2 A Backjumping KRUSTTool

In refinement, we incrementally refine the KBS to correctly answer the current and previous refinement examples. So, the most natural analogy between CSPs and refinement links variables with training examples, the current variable with the refinement example, and instantiated variables with correctly solved constraint examples in cebuf. CSP constraints correspond to achieving consistency with the constraint examples. Finally the domain for a variable is the repairs triggered by fault evidence provided by the refinement example.

The KRUSTTool algorithm must backjump when the refinement example e_i and the input KBS R fail to create any refined KBSs (i.e. the generated KBSs $Generated_{Ri}$ is empty) or those generated are rejected by the constraint examples (i.e. the filtered KBSs $Filtered_{Ri}$ is empty). If $Filtered_{Ri} = \{\}$ then we must determine the most recent constraint example that caused the removal of each generated KBS, and backjump there. If $Generated_{Ri} = \{\}$ then BT is the only option; no conflicting constraint examples can be identified since there are no KBSs to test!

Let us revisit the scenario in Figure 2. $Generated_{R_{415}} = \{R_{51}, R_{52}, R_{53}, R_{54}\}$ and so refinement can backjump, but since $Filtered_{R_{415}} = \{\}$, each KBS in $Generated_{R_{415}}$ must have been rejected by at least one constraint example in cebuf. Suppose R_{51}, R_{52} wrongly solve e_2, and R_{53}, R_{54} wrongly solve e_3. Then refinement will continue from e_3, because it is the most recent on the path. The next available refined KBS R_{32} is selected with e_4 as the refinement example. Finally, e_5 is moved back into tebuf to be a future refinement example. If no more KBSs were available from e_3 then BJ backtracks to node e_2.

There are two obvious differences between CSPs and refinement. Firstly, the domain of potential repairs is not known in advance, instead it is constructed incrementally during refinement generation and filtering. This is handled by associating refined KBSs with refinement examples that generate them, and reasoning about backjumping using constraint examples rather than KBSs. Secondly, the behaviour of constraint examples can change – they can become uncorrected and so provide new fault evidence generating further refined KBSs. The algorithm identifies and reinstates these examples in tebuf.

4 Improving Backjumping for Refinement

Backjumping was introduced as a way to reduce the search of backtracking. Contrary to expectation we found that BJ often increases the number of iterations but that these extra iterations were used profitably and the BJ KRUSTTool on average provided refined KBSs with lower error-rates than the BT KRUSTTool [18]. Therefore our next concern is to improve efficiency of the BJ KRUSTTool by reducing the number of refinement cycles without increasing the error-rate. We investigated techniques that improve CSP search efficiency [15].

- Value ordering heuristics select those values that conflict least with variables that are yet to be instantiated;

- Variable ordering heuristics deal with most constrained variables first.

CSP value ordering is analogous to ordering the refined KBSs; KRUSTTools already does this when the accuracy filter orders the refined KBSs in increasing order of error-rate on tebuf. In fact the general KRUSTTool approach is closely related to the repair-based approach to solving CSPs and its greedy min-conflict heuristic for repair selection [11]; and the refined KBS ordering itself is similar to the look-ahead value ordering min-conflicts heuristic that ranks the values of a variable in increasing order based on the number of incompatibilities with values of future variables [6].

For the rest of this paper, we concentrate on how variable ordering can be applied to a KRUSTTool. A CSP variable is generally constrained in two ways, firstly by the constraints it is involved in and secondly by its domain size. Most common variable ordering heuristics exploit these 2 properties [5, 8]. Heuristics for static ordering exploit relationships among variables identified from the topology of the constraint graph [17]. Dynamic variable ordering addresses the fact that invariably the best variable order is different in different branches of the search tree, by taking advantage of the information available after each variable instantiation to move the search to branches that are more likely to contain a solution [9]. Various look-ahead strategies select the variable that most constrains the remainder of the search [16]. The motivation behind all such heuristics is to deal with difficult variables first.

We now turn to how this is applied to knowledge refinement. CSP variables involved in the most or tightest constraints correspond to training examples whose repairs put the highest consistency demands on other training examples; current work investigates clustering training examples as a way to address this. CSP variables with the smallest domain correspond to refinement examples that generate the smallest set of refined KBSs in a refinement cycle. But, going as far as refinement generation can be computationally expensive. In this paper we establish heuristics that predict how constrained the refinement cycle for each training example will be, and use these to order the training examples.

4.1 Evidence From the Recent Refinement Cycle

Simple constrainedness information comes from the newly completed refinement cycle; where the final step executed all the refined KBSs generated in that cycle on the remaining training examples in tebuf. Although this was done to calculate the error-rate of each of these refined KBSs, it also determines an estimate of how faulty each training example is; i.e. how many of these refined KBSs got the training example wrong. Remember, all these refined KBSs are related since they were all derived from the previous best refined KBS.

Table 1 demonstrates how fault evidence from the most recent refinement cycle can be employed to select the next refinement example from tebuf. Let us assume that m refined KBSs $R_{i1}, R_{i2}, \ldots, R_{im}$ were generated with e_i as the refinement example and that tebuf now contains training examples e_{i+1},

$e_{i+2},\ldots,e_n$. The table entry for e_j and R_{ik} has value 1 if R_{ik} **incorrectly** answers e_j, and 0 otherwise. Therefore, the *error-rate* of R_{ik} on tebuf, $err_{R_{ik}}$, is the column total divided by n. The row total f_j is the level of *faultiness* of e_j as judged by $R_{i1}, R_{i2}, \ldots, R_{im}$. The refined KBS with the lowest error-rate, $\min(err_{R_{ik}})$, is selected as the best refined KBS. We now use, the training example with the highest level of faultiness, $\max(f_j)$, as the next refinement example. All ties are broken randomly.

Generated Refined KBSs

tebuf	R_{i1}	R_{i2}	...	R_{im}	*faultiness*
e_{i+1}	1	0	...	0	$f_{e_{i+1}}$
e_{i+2}	0	1	...	0	$f_{e_{i+2}}$
$\vdots$	$\vdots$	$\vdots$	...	$\vdots$	$\vdots$
e_n	1	1	...	0	f_{e_n}
error-rate	$err_{R_{i1}}$	$err_{R_{i2}}$	...	$err_{R_{im}}$	

Table 1: Faultiness of remaining examples.

This heuristic is reminiscent of the best known CSP dynamic ordering heuristic, dynamic search rearrangement (DSR), which selects the next variable having the minimal number of values that are consistent with the current partial solution [5]. Of course the difference is that with knowledge refinement the set of potential refined KBSs is not known in advance and so we use fault evidence from the most recent potential refined KBSs as the basis for selecting the most constrained training example for the next iteration.

4.2 Evidence From How the Problem was Solved

A more direct estimate of how many refinements will be generated for a particular training example is the number of places where the problem solving behaviour for that training example can be changed. The KRUSTTool algorithm already creates a data structure containing precisely this information. The *problem graph* captures the problem-solving for the refinement example and allows the KRUSTTool to reason about the fault that is being demonstrated [4]. Essentially, the problem graph records what happened, and also shows all possible rule activation routes to the required goal, of which only one is actually used. Problem graphs can become quite complex with long chains and complicated branching. Figure 3 shows some simple problem graphs with which we illustrate their function. Training example A is a problem described by a set of observables including $T_{A1} - T_{A4}$, which the expert solves as $goal_A$. However the KBS currently reasons from the observables by applying leaf rules $R7$ and $R4$, which together allow a middle rule $R13$ to fire, and finally the end rule $R11$ concludes S_A the faulty solution. The darkened area of the problem graph is the *positive problem graph* and corresponds to the problem solving that has been undertaken by the faulty KBS. Therefore it contains the solution sub-

graph for the training example but also contains other partial proofs; e.g. T_{A1} allows $R7$ to fire, but this only partially satisfies $R12$. The positive problem graphs for the other 2 training examples are similar but notice neither provides a solution since each partial solution subgraph terminates with an intermediate result.

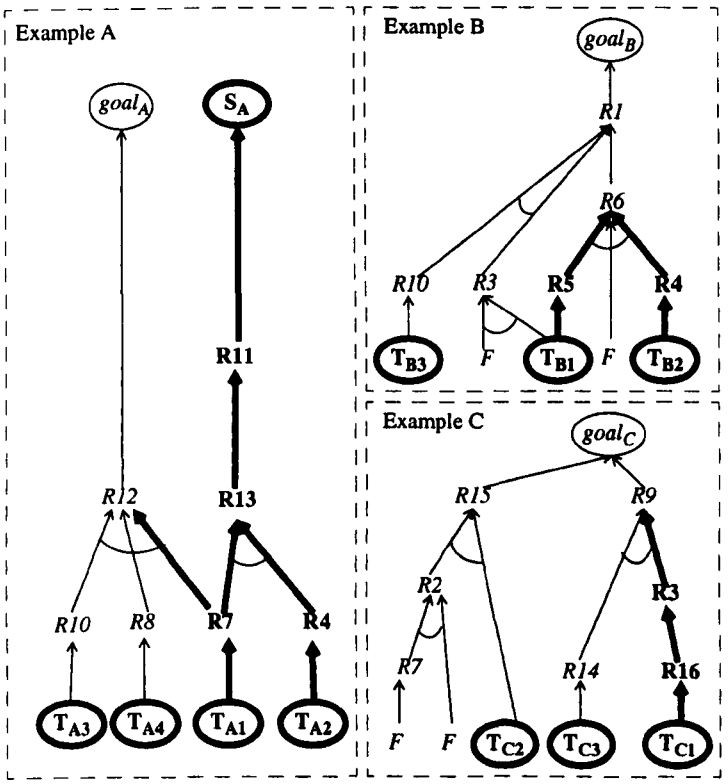


Figure 3: Problem Graph for training examples, A, B and C.

Repairs correspond to preventing faulty rule chains from being activated and so the number of rule activations in the positive problem graph is a simple measure to predict the number of potential refined KBSs, and hence how constrained the refinement cycle for that training example will be. Activation counts for training examples A, B and C in Figure 3 return 4, 2 and 2 respectively, indicating that B and C are the most (and equally) constrained and so will be selected over A. All ties are broken randomly.

4.3 Evidence From How the Problem Should be Solved

The problem graph captures more about the problem-solving than simply recording what happened. It also contains a ***negative problem graph*** that shows all possible rule activation routes to the required goal. Thus in Figure 3 the

expert's solution for training example A ($goal_A$) has not been proved because, $R10$, $R8$ and $R12$ are only partially satisfied, and are unable to fire. We have not darkened the arrows leading from T_{A3} and T_{A4} to indicate that the conditions in $R10$ and $R8$ do not match observables T_{A3} and T_{A4}, and must be weakened before they are satisfied. In contrast conditions in $R4$, $R13$ or $R11$ must be strengthened in order to stop S_A being asserted. Similar explanations hold for training examples B and C, but now in addition some rule conditions (e.g. the first condition of $R3$ for training example B), cannot be weakened to match any observable or rule conclusion and so are not linked to any rule or observable but instead these "non-observables" are labelled F.

The negative problem graph provides additional information on how constrained the refinement cycle will be. Counting all the rule "activations" in both the positive and negative parts of the problem graph provides a second measure of constrainedness. This measure promises to be more informative since it adds the locations of possible repairs in the negative problem graph to those from the positive part.

In practice we found it was better to distinguish between rules in the negative problem graph whose conditions could be weakened to match observables from those that could never match. We amended the heuristic so that it ignored any negative rule activation whose conditions are *all* linked to (or derived from) "non-observables" (F's in the diagram); e.g. rules $R7$ and $R2$ will be omitted from C's count. Without this modification the heuristic can estimate a training example like C to be less constrained than it actually is. Such an amendment requires the assumption that training examples are noise free and that leaf rules are correct, however this seems acceptable given our need simply to estimate constrainedness.

Table 2 lists all the refinement places for the 3 training examples at the left. The count of rule activations in the complete problem graph, with and without the non-observables correction, appears at the right. Therefore, example C with the lowest improved rule activation count is selected over A and B. We note that although the improved heuristic is a good predictor of the number of refinements here, more complex problem graphs may need a more sophisticated way to combine rule activation counts from the positive and negative parts of the problem graph.

5 Results

We evaluate backjumping KRUSTTools that apply static and/or dynamic ordering of the training examples using the heuristics we have developed in Section 4. The problem graph heuristics define a static ordering of the training examples before the iterative refinement cycles are started. They can also be used for dynamic ordering where the measures are recalculated on the best refined KBS output from a cycle and applied to re-order the remaining training examples. The emphasis of the evaluation is to compare the number of iterations, error-rate and finally the resource usage.

Training Example	*Refinements*				*Rule Activations*	
	Strengthen	*Weaken*	*None*	*Count*	*All*	*Improved*
A	*R*4 *R*13 *R*11	*R*10 *R*8 *R*12	*R*7	6	7	7
B		*R*1 *R*3 *R*10 *R*6	*R*5 *R*4	4	6	6
C		*R*9 *R*14 *R*15	*R*3 *R*16 *R*7 *R*2	3	7	5

Table 2: Refinements and rule activations from the complete problem graph.

Our testbed is a corrupted student loans KBS, created by introducing 5 faults to the 20 rules in the original KBS [13]: an extra rule, a changed comparison operator in 2 rules and an extra condition in 2 other rules. Although this is not a highly realistic scenario, the faults are sufficiently interacting that it allows experimentation in carefully controlled conditions.

Since our experiments involve an assessment of the effectiveness of backjumping with various orderings of training examples, we had to ensure that backtracking is triggered. We chose 8 specific "difficult" examples from the standard student loans dataset that are correctly answered by the uncorrupted KBS, but whose repairs for the corrupted KBS are particularly conflicting. In fact there are 9 ways to pair these 8 examples so that the refined KBSs triggered by one training example tightly interacts with the other's refined KBSs. We then randomly selected a further 22 "normal" examples to make a 30 example dataset for our experiments.

For each run we randomly select n conflicting pairs, duplicates are removed and further examples are randomly selected (from the "normal" examples) until the training set contained 15 examples. The remaining 15 examples become the independent test set. The dataset was partitioned this way 20 times, with 8 conflicting pairs in the first 10 runs, and 9 conflicting pairs in the next 10 runs. The results of each experiment refer to these 20 training/test splits. Significance results are based on a 95% confidence level and apply the Wilcoxen signed-rank test (2 data sets) or the Kruskal Wallis test (3 or more data sets), since our data is not normally distributed.

5.1 Static Ordering

Static ordering provides a sequence of training examples prior to the iterative refinement cycles. We compare two orderings using the problem graph heuris-

tics[1] with a random ordering.

- RANDOM: move all correctly solved training examples into cebuf then randomly order tebuf.

- PGRAPH+: move all correctly solved training examples into cebuf, then sort the remaining training examples in decreasing order of the number of rule activations in the positive problem graph only.

- PGRAPH±: as for PGRAPH+ but use the number of rule activations in the complete problem graph (positive and negative) including the modification for "non-observables".

Error-rate for the final refined KBS was not impaired by PGRAPH+ and PGRAPH±, and they both reduced the error compared to RANDOM in 4 test runs. More pertinent to this evaluation is the number of iterations for these three algorithms listed in Table 3. PGRAPH+ required significantly (p-value = 0.028) fewer iterations compared to RANDOM; 10 test runs had fewer iterations and only 2 test runs had more iterations and this was at most 2 iterations longer. PGRAPH± improved on PGRAPH+ by reducing the number of iterations in 4 test runs, however despite the added information acquired from the negative problem graph this reduction is not statistically significant. Any improvements in PGRAPH± over PGRAPH+ is due to the added information causing fewer ties, which essentially mean fewer randomly resolved tie-breaks. This may be explained by observing that refinement generation explores both the positive and negative problem graphs and that refinements can include changes to both parts of the reasoning. Therefore a more complex combination of the rule activation counts may be required so that it takes account of those activations that contribute towards the required goal and are also part of the positive problem graph, by not counting them as individual activations.

Static ordering	*Mean*	*Median*	95% *Confidence*
RANDOM	9.05	8.0	±1.420
PGRAPH+	7.65	7.0	±0.717
PGRAPH±	7.65	7.5	±0.410

Table 3: Number of iterations for static ordering.

The test results clearly indicate that the order in which training examples are processed by the KRUSTTool affects the number of backjumps and iterations. It also confirms that the number of rule activations is an indicator of the level of constraint of a training example.

[1]The other heuristic (Section 4.1) can only be applied as a dynamic ordering since it exploits information from all the refined KBSs from the previous cycle.

5.2 Dynamic Ordering

The original backjumping KRUSTTool already employs one form of dynamic ordering by reinstating latent examples; these are constraint examples that did not require refinement at the time, and so contributed no fault evidence as refinement examples, but are now incorrectly solved by the current KBS and so are moved back into tebuf. This reordering is applicable only when backjumping occurs. We now extend training example ordering by applying each of the three heuristics from Section 4 to also reorder before every refinement cycle, where again ties are ranked randomly. This more general reordering is employed first, to ensure that reordering enforced by backjumping is not undone.

1. Current best refined KBS is the input faulty KBS.
2. Apply static ordering on tebuf.
3. Loop until tebuf is empty:
   (a) Execute the refinement cycle with the current best refined KBS and the top most example in tebuf to generate and filter the refined KBSs.
   (b) Apply dynamic ordering on tebuf.
   (c) If the set of filtered refined KBSs is not empty then choose the current best refined KBS.
   (d) If the set of filtered refined KBSs is empty:
      i. If there are latent examples then these are pushed into tebuf, after all correctly solved training examples are moved into cebuf.
      ii. Otherwise, employ BJ to identify the conflict example and its next best refined KBS to backtrack to, and all constraint examples on the way are moved back into tebuf.

Figure 4: Algorithm combining static and dynamic ordering.

Figure 4 outlines the basic algorithm combining static and dynamic ordering in a BJ KRUSTTool algorithm. Any of the three static orderings RANDOM, PGRAPH+, PGRAPH± from Section 5.1 can be used in step 2 and influences the selection of the first refinement example only. Dynamic ordering occurs in step 3b, where any of the following can be applied:

- FAULTBASED: re-order tebuf in decreasing order according to evidence from KBSs from the recent refinement cycle (Section 4.1), after moving all correctly solved training examples from tebuf into cebuf; or
- DYNPGRAPH+: apply PGRAPH+'s heuristic (now in every cycle); or
- DYNPGRAPH±: apply PGRAPH±'s heuristic (now in every cycle).

5.3 Static and Dynamic Combinations

Our experiments looked at seven (of the nine possible) static-dynamic combinations; we used the same problem graph heuristic in the static and dynamic orderings. Once again the error-rate of the final KBS was unaffected. Comparing the results in Table 4 with the static ordering results in Table 3, we see that all combinations have reduced the number of iterations by at least two iterations. All heuristics employing the complete problem graph resulted in lower average number of iterations but FAULTBASED results are very close. However the differences among all the static + dynamic combinations are not significant; PGRAPH± + DYNPGRAPH± has the fewest iterations but this is not significant ($p = 0.932 > 0.05$). These results show that using static + dynamic ordering gives significant gain over using static ordering only but that none of the combinations is better than any other.

Static	+	*Dynamic*	*Mean*	*Median*	95% *Confidence*
RANDOM	+	FAULTBASED	5.15	5	±0.532
RANDOM	+	DYNPGRAPH+	5.40	5	±0.765
RANDOM	+	DYNPGRAPH±	5.15	5	±0.613
PGRAPH+	+	FAULTBASED	5.60	5	±0.864
PGRAPH+	+	DYNPGRAPH+	5.80	5	±0.893
PGRAPH±	+	FAULTBASED	5.10	5	±0.524
PGRAPH±	+	DYNPGRAPH±	5.05	5	±0.557

Table 4: Number of iterations for static+dynamic ordering combinations.

We have succeeded in reducing the number of iterations but at what computational cost? Table 5 shows the number of cpu cycles for our seven heuristic combinations; the figures for static ordering only have been included for reference. FAULTBASED has proved to be very effective for dynamic ordering since the overhead of applying it with any static ordering is not significant. The orderings based on problem graphs have not been so effective; any gain in reducing the iterations has been overwhelmed by the expense of each iteration. We hope that with more complex KBSs, the richness of the information in the problem graph will result in sufficient quality gains in the refined KBS that the expensive computation is worthwhile.

		Static		
		RANDOM	PGRAPH+	PGRAPH±
Dynamic	None	286480	453030	384910
	FAULTBASED	246060	454590	398670
	DYNPGRAPH+	477760	564810	
	DYNPGRAPH±	581020		798910

Table 5: Cpu cycles for static + dynamic combinations.

The reduction in the number of iterations may actually be worthwhile, even

at the expense of some increase in the total effort. Many iterations to achieve consistency with a training set may be regarded as many tinkering repairs; while fewer more fundamental repairs may create a higher quality KBS.

6 Conclusions

The emphasis of this paper is improving the search efficiency of backtracking KRUSTTools, and in particular BJ KRUSTTools, however this approach is applicable more generally. Refinement algorithms tend to use a hill-climbing approach, and so to avoid suboptimal refined KBSs, they should introduce some form of backtracking, and thus could benefit from backjumping.

BJ KRUSTTools produce final refined KBSs with lower error-rates than BT KRUSTTools since the repairs for potentially conflicting training examples are often handled in consecutive cycles, leading to repairs that are better for new problems. However, despite the fact that backjumping had been introduced as a more informed search than chronological backtracking, BJ KRUSTTools result in more iterations. This paper explored methods to reorder training examples with the goal of improving BJ KRUSTTools by reducing the number of iterations whilst maintaining the accuracy of the final refined KBS.

Two static orderings were defined from two heuristics based on counting rule activations. Both maintained the reduced error-rates of backjumping with no example ordering as reported in [18] but achieved this in fewer iterations. The information from the negative problem graph allowed PGRAPH± to cause fewer tie-breaks. Further work could investigate how the heuristics can be extended to resolve tie-breaks strategically as opposed to randomly as at present. We also believe that the overlapping rule activations from the positive and negative problem subgraphs should be exploited to give a more informed heuristic for PGRAPH± and DYNPGRAPH±.

Three dynamic orderings were defined by these two heuristics and a simpler fault evidence heuristic. Algorithms combining static and dynamic ordering further reduced the number of refinement cycles, without increasing the error-rate of the final refined KBS. An important issue with dynamic ordering is the additional computational effort introduced by the reordering at each cycle. FAULTBASED very effectively guided the search without adding much computation and for one combination actually lowered the total effort, but the problem graph heuristics were computationally very expensive. However, we are currently reordering the complete set of remaining training examples from scratch every cycle. Future work will investigate whether knowledge about the repair from the previous cycle will allow less frequent calculation of the problem graph heuristics, or a more targetted application to examples that are most likely to be highly constrained. The calculation for the previous cycle or knowledge of the repair may provide a suitable estimate or an incremental update of the value for this cycle. More experience of the effect of re-ordering may limit the number of training examples that need to be considered. Current work on clustering training examples may also focus the reordering effort.

We must bear in mind that our search space is extremely dynamic with sequences of refinement examples altering the refined KBSs being considered. As with CSPs, our goal is to reduce the search effort and still find a good sequence of repairs rather than simply hill-climb through the repair space without backtracking. But unlike CSPs, where an instantiation for one variable can only restrict the domain of another, in knowledge refinement the repair for one training example may also lead to a totally different set of proposed refinements for a later training example.

Acknowledgments

The KRUSTWorks project is supported by EPSRC grant GR/L38387 awarded to Susan Craw. Nirmalie Wiratunga is partially funded by ORS grant 98131005.

References

[1] J. R. Bitner and E. Reingold. Backtrack programming techniques. *Communications of the ACM*, 18:651–656, 1975.

[2] C. Blake and E. Keogh and C.J. Merz. UCI Repository of Machine Learning Databases. In University of California, Irvine, Dept. of Information and Computer Sciences, 1998. www.ics.uci.edu/~mlearn/MLRepository.html.

[3] Robin Boswell, Susan Craw, and Ray Rowe. Knowledge refinement for a design system. In Enric Plaza and Richard Benjamins, editors, *Proceedings of the Tenth European Knowledge Acquisition Workshop*, pages 49–64, Sant Feliu de Guixols, Spain, 1997. Springer.

[4] Susan Craw and Robin Boswell. Representing problem-solving for knowledge refinement. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pages 227–234, Menlo Park, California, 1999. AAAI Press.

[5] Rina Dechter and Itay Meiri. Experimental evaluation of preprocessing algorithms for constraint satisfaction problems. *Artificial Intelligence*, 68:211–341, 1994.

[6] Daniel Frost and Rina Dechter. Look-ahead value ordering for constraint satisfaction problems. In *Proceedings of the Fourteenth IJCAI Conference*, pages 572–578, 1995.

[7] J. Gaschnig. Performance measurements and analysis of certain search algorithms. Technical Report CMU-CS-79-124, Carnegie-Mellon University, PA, 1979.

[8] Ian Gent, Ewan MacIntyre, Patrick Prosser, Barbara Smith, and Toby Walsh. An empirical study of dynamic variable ordering heuristics for the

constraint satisfaction problem. In *in Principles and Practice of Constraint Programming*, pages 179–193. Springer-Verlag, 1996.

[9] R.M. Haralick and G.L. Elliott. Increasing tree-search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[10] Grzegorz Kondrak and Peter van Beek. A theoretical evaluation of selected backtracking algorithms. *Artificial Intelligence*, 89:365–387, 1997.

[11] Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird. Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–205, 1992.

[12] D. Ourston and R. Mooney. Theory refinement combining analytical and empirical methods. *Artificial Intelligence*, 66:273–309, 1994.

[13] Michael J. Pazzani. Student loan relational domain. In UCI Repository of Machine Learning Databases [2], 1993.

[14] B. Richards and R. Mooney. Automated refinement of first-order horn-clause domain theories. *Machine Learning*, 19:95–131, 1995.

[15] Norman M. Sadeh and Mark S. Fox. Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem. In *Proceedings of the Fourth International Conference on Expert Systems in Production and Operations Management*, pages 134–144, 1990.

[16] Barbara Smith and Stuart Grant. Trying harder to fail first. In *Proceedings of the ECAI98 Conference*, pages 249–253, Brighton, UK, 1998. John Wiley and Sons Ltd.

[17] Edward Tsang. *Foundations of Constraint Satisfaction.* Academic Press, San Diego, 1993.

[18] Nirmalie Wiratunga and Susan Craw. Incorporating backtracking search with knowledge refinement. In Anca Vermesan and Frans Coenen, editors, *Proceedings of the Sixth European Symposium on the Validation and Verification of Knowledge Based Systems*, Oslo, Norway, 1999. Kluwer Academic Publishers.

Appears in Proceedings of the 12th International Conference on Knowledge Engineering and Knowledge Management (EKAW2000)

Informed Selection of Training Examples for Knowledge Refinement

Nirmalie Wiratunga and Susan Craw

School of Computer and Mathematical Sciences
The Robert Gordon University
St Andrew Street, Aberdeen AB25 1HG
Scotland, UK.

Email: nw|s.craw@scms.rgu.ac.uk

Abstract. Knowledge refinement tools rely on a representative set of training examples to identify and repair faults in a knowledge based system (KBS). In real environments it is often difficult to obtain a large set of examples since each problem-solving task must be labelled with the expert's solution. However, it is often somewhat easier to generate unlabelled tasks that cover the expertise of a KBS. This paper investigates ways to select a suitable sample from a set of unlabelled problem-solving tasks, so that only the subset requires to be labelled. The unlabelled examples are clustered according to the way they are solved by the KBS and selection is targeted on these clusters. Experiments in two domains showed that selective sampling reduced the number of training examples used for refinement, and hence requiring to be labelled. Moreover, this reduction was possible without affecting the accuracy of the final refined KBS. A single example selected randomly from each cluster was effective in one domain, but the other required a more informed selection that takes account of potentially conflicting repairs.

1 Introduction

Knowledge refinement is incremental learning, where the learning must adapt existing knowledge in a Knowledge-Based System (KBS). Refinement tools aid knowledge engineers by assisting with the knowledge debugging and maintenance phases in the Knowledge-Based Systems development cycle [1–3]. These tools ensure that the KBS's solution is consistent with that of a domain expert for a given task. In common with other learning algorithms, the tasks and the expert's solutions are maintained as training examples. Refinement is triggered when the system's and expert's solution for a given task are inconsistent. Although training examples that indicate faults are useful to drive refinement, access to correctly

solved training examples is beneficial, because, they help focus refinement by ensuring that repairs are not too closely fitted to wrongly-solved examples.

The choice of training examples for refinement becomes important when one of the constraints on the refinement process is a limited number of labelled training examples. This is a relatively common problem in a real environment, where labelling many problem-solving tasks with the expert's solution may require significant interaction with a busy expert. Unlabelled training examples are often generated by using domain knowledge already embodied in the KBS or meta-knowledge [4]. Therefore, unlike the labelling task, generating unlabelled examples does not typically require the expert. The goal of the work described in this paper is to perform an informed selection from a set of unlabelled training examples which the expert must subsequently label, thereby reducing the demand on the expert. However, we must ensure that the informed selection of relevant training examples does not hamper the refinement process by omitting examples that uniquely reveal faults.

The problem of unavailability of labelled training examples and sample selection of relevant examples from a set of unlabeled examples falls under the paradigm of active learning and more specifically, selective sampling. Much work has been done in selective sampling mainly related to training classifiers: for nearest neighbour, using a lookahead approach that selects examples based on statistical information about the utility of the resulting classifier [5]; for text classification, using a committee-based approach combined with expectation maximization [6]; and for C4.5 using a probabilistic classifier that selects examples based on class uncertainty [7]. Increasingly, estimation and prediction techniques with roots in statistics are being applied to classifiers with improved accuracy results [8]. However, the use of examples for training classifiers differs from their use for refinement tools:

- in refinement, examples are used to expose faults in an existing KBS and so are employed to refine incomplete concepts and not learn from scratch; and
- examples are used for refining KBSs that model, not only classification tasks but also design tasks [9] and even planning tasks [10].

Direct application of currently available selective sampling methods for learning classifiers to refinement tools is therefore, not straightforward. We adopt the common approach of partitioning the available examples into clusters, but exploit the relationship between the examples and how they are solved by the faulty KBS, in contrast to existing selection techniques that exploit the statistical distribution of examples. As a result our clusters will contain examples that trigger similar problem solving behaviour in the KBS. We then apply various heuristics that help select examples from clusters. However, the presence of interacting faults in a KBS complicates sample selection since they require the selection of more than one example from each cluster. We have developed heuristics that identify those examples that are most likely to demonstrate interacting faults and we propose algorithms that apply these heuristics to example selection. The selected subset of examples is then presented to the expert for

labelling. Once labelled, these examples can be used by the refinement tool to drive the refinement process.

Section 2 introduces iterative refinement by describing the process undertaken by a particular family of refinement tools. The selective sampling process in Section 3, firstly, describes a structure that captures problem-solving behaviour of the KBS for a given task, secondly, presents a clustering framework that uses this problem-solving behaviour to determine similarity between unlabelled examples, and thirdly, identifies several heuristics for selecting a suitable number of examples from these clusters. Experimental results from evaluating the selection heuristics on two problem domains which have different problem-solving characteristics is presented in Section 4. Finally, in Sections 5 and 6, we discuss related work in the field and conclude with contributions of this work and implications for future work.

2 Refinement with KRUSTtools

The KRUSTWORKS project has developed a generic knowledge refinement framework. Given a specific rule-base shell, this framework is used to generate a refinement tool, a KRUSTtool, by re-using core refinement modules. These modules are applied to generic knowledge structures which model the behaviour of the rule-base. The structures are formed by translators that work on the specific rules and the associated traces [1]. The currently developed framework is able to deal with faulty KBSs implemented in shells incorporating reasoning strategies that can be forward-chaining, backward-chaining or both.

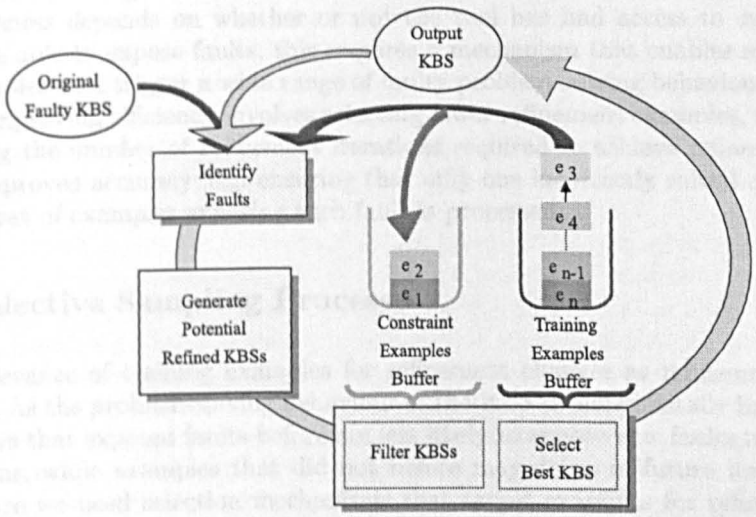


Fig. 1. The KRUSTtool Process.

In common with many refinement tools, KRUSTtools incrementally refine a KBS based on fault evidence provided by labelled training examples. A labelled training example e is a task-solution pair $\langle [f_1, \ldots, f_m], goal \rangle$; the observables $f_1, \ldots, f_m$ are the facts that initialise the problem-solving task, and its solution *goal* is the example's label acquired from the expert. The KRUSTtool's refinement process is iterative with labelled training examples $e_1, \ldots, e_n$, utilized one at a time (Figure 1). The input KBS for each iteration is the best refined output KBS from the previous iteration, or the original faulty KBS in the first iteration. The training examples buffer contains all labelled examples that are yet to be used by the KRUSTtool. For each iteration, the top example in this buffer is chosen as the refinement example and drives that refinement cycle. If the refinement example is correctly solved by the input KBS then refinement is not required, otherwise the fault evidence is employed to allocate blame. The refinement algorithm then identifies various ways by which the required target solution can be attained and generates several potential refinements and implements them as refined KBSs. Once used, the refinement example is then transferred into the constraint examples buffer, which is simply the buffer that keeps track of examples previously solved by the KRUSTtool. However, an important task of this buffer is to help filter refined KBSs, by rejecting those that incorrectly answer any of the examples in it. The filtered refined KBSs are then ranked by their accuracy on the training examples buffer, and the refined KBS with the highest accuracy is the output KBS for this iteration.

Fundamental to the KRUSTtool's successful refinement operation is the availability of labelled examples for its buffers. Availability is often constrained by limited expert interaction and high processing costs. The KRUSTtool should ideally be able to handle such situations by actively selecting training examples from an available set of unlabelled examples. Selected examples must be beneficial for improving the effectiveness and efficiency of the refinement tool. The effectiveness depends on whether or not the tool has had access to examples that are able to expose faults; this requires a mechanism that enables selection of examples that trigger a wide range of faulty problem-solving behaviour in the KBS. Improving efficiency involves selecting fewer refinement examples, thereby reducing the number of refinement iterations required to achieve refined KBSs with improved accuracy; e.g. ensuring that only one incorrectly solved example from a set of examples exposing each fault is processed.

3 Selective Sampling Process

The relevance of training examples for refinement changes as refinement progresses. As the problem-solving behaviour of the KBS is incrementally improved examples that exposed faults before are less likely to expose new faults in future iterations, while examples that did not before may do so in future iterations. Therefore we need selection mechanisms that target examples for refining the KBS given its current problem-solving behaviour. The use of selective sampling for the KRUSTtool encompasses an informed selection of examples, the labelling

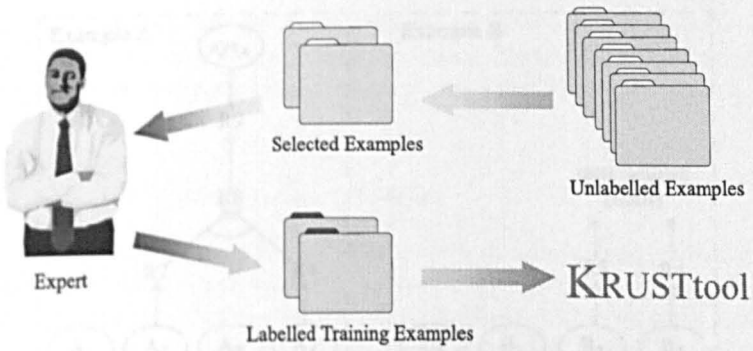


Fig. 2. A single iteration of *select-label-refine*.

of these selected examples by the expert, and the refinement of the faulty KBS using the batch of labelled examples. A single iteration of this *select-label-refine* process provides a small batch of labelled examples to use as the initial training examples buffer for the KRUSTtool (see Figure 2). Once the KRUSTtool has incrementally refined the KBS to correctly solve these labelled examples, the next iteration of *select-label-refine* can be triggered. In practice, the *select-label-refine* process must be repeated until; no further faults are exposed in the KBS hence, no improvement in accuracy; or a limit on the number of examples an expert is willing to label is reached.

3.1 Problem-solving Behaviour

The KRUSTtool records the problem-solving that is undertaken by a KBS for an example in a structure, the *positive problem graph*[1]. Essentially it records the rule activations and the order in which these activations occur. Figure 3, shows two simple positive problem graphs for a fictitious faulty KBS with rules R_4, R_5, R_7, R_8, R_9 among others. Let us assume that each positive problem graph captures the observed problem solving behaviour of the faulty KBS, as a result of executing each of the two unlabelled examples, A=$\langle[A_1, \ldots, A_4], ?\rangle$ and B=$\langle[B_1, \ldots, B_4], ?\rangle$. With example A, the KBS reasons from the observables by applying leaf rules R_7 and R_4, which together allow a middle rule R_8 to fire, and finally the conclusion of the end rule R_9 provides the system solution, sys_A. A system solution would typically be a class, a design, a formulation or a plan, depending on the type of problem domain. Similar explanations hold for example B's positive problem graph, but notice that reasoning has not progressed beyond the conclusions of leaf rules, R_4 and R_5. Here, we have an intermediate result but no obvious system solution. We shall use the similarity between the positive problem graphs of examples to determine which examples may indicate the same faults in the KBS. The task of establishing similarity in this manner means that we need only be interested in rule activations for examples, regardless of whether or not the system solution is correct. Therefore, more importantly, examples need not be labelled for this task.

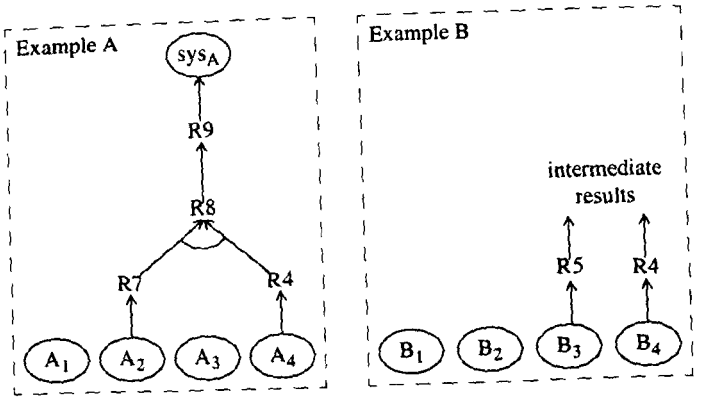


Fig. 3. Positive Problem Graphs for examples A and B.

3.2 Cluster Formation

To form example clusters we need to define a *similarity metric* which is then utilized by a *clustering technique* that progressively develops the clusters. Since examples are presented as a vector of observables, an obvious similarity metric compares these vectors. However, in knowledge refinement we are interested in sampling examples with respect to problem-solving behaviour of the faulty KBS and so our similarity metric reflects this by making use of the positive problem graph. Given a KBS containing rules $R_1, \ldots, R_N$, we define a binary valued *rule vector* corresponding to an example e as $\mathbf{r} = (r_1, \ldots, r_N)$, where $r_i = 1$ if R_i appears in the problem graph for e; and $r_i = 0$ otherwise. Thus, the rule vector for the training example A in Figure 3 is (0, 0, 0, 1, 0, 0, 1, 1, 1, 0), where N=10. Here the 1's correspond to rule activations $R4$,$R7$,$R8$ and $R9$.

The similarity measure needs to capture refinement similarity between two unlabelled training examples e_1, e_2. As refinement similarity depends on the similarity in problem solving behaviour, the similarity between e_1, e_2, can be established by comparing their rule vectors $\mathbf{r}_1, \mathbf{r}_2$. For this purpose the Euclidean distance metric may be used, but it can lead to two rule vectors being regarded as highly similar despite them having no common rule activations. Association coefficients [11] avoid this by focusing on the common rule activations and normalizing by the number of rule activations in both rule vectors, thereby ignoring rules that are not activated. We employ the Dice coefficient, a commonly used similarity measure of this type:

$$RefSim(e_1, e_2) = Dice(\mathbf{r}_1, \mathbf{r}_2) = \frac{2\ \mathbf{r}_1 . \mathbf{r}_2}{\mathbf{r}_1 . \mathbf{r}_1 + \mathbf{r}_2 . \mathbf{r}_2}$$

We then use an agglomerative hierarchical clustering technique, where training examples with the greatest similarity are united in small clusters and these clusters are iteratively fused until intra-cluster similarity achieves a predetermined threshold. The decision to fuse clusters is based on the farthest neighbour

principle [12], where those two clusters that have the minimum distance between their most dissimilar cluster members are fused. Typically, this form of cluster fusion leads to small, tightly bound clusters, provided that the fusion threshold is low.

3.3 Selecting Examples using Clusters

Clusters provide information that allows a more informed choice than a random selection of examples. Each cluster represents the problem-solving behaviour pertaining to some part of the faulty KBS, because examples with similar rule activations are clustered together. If we happen to know which area of the KBS is faulty, the task of example selection is reduced to picking the cluster related to that area. However, in most cases the KRUSTtool has no prior knowledge about what parts of the KBS might be faulty, and so we need a more general selection technique that targets all potentially faulty parts of the KBS.

Since each cluster contains examples which are solved in a similar way by the KBS, it might appear reasonable to assume that repairing a fault exposed by a single example from a cluster would correct the rest of the cluster. One selection method CLUSTERREP exploits this assumption by randomly selecting one example from each cluster. Certainly, training examples that activate several rules in common appear in the same cluster and typically are also similar in their observables. However, in some situations examples from a single cluster may not have similar observables, and so may contain a pair of examples where a possible repair for one example introduces a fault into the repaired solution for the other; or result in no obvious repair. Faults of this nature are termed *interacting faults* and the involved pair of examples is termed a *conflict pair*.

3.4 Faults that Interact

To demonstrate the effects of interacting faults on refinement we use 4 Clips rules taken from a corrupted version of a student loans adviser. Of these rules, two have been corrupted by adding extra conditions, highlighted in bold (see Figure 4). Here, $R16$ translates to "if a student has filed for bankruptcy and is enlisted then grant the student a financial deferment", and $R19$ translates to "if a student is disabled and has filed for bankruptcy then grant the student a disability deferment". Assume that the KRUSTtool is attempting to fix these rules based on fault evidence provided by training example x and y in that order.

$$x = \langle [(\texttt{filed\_for\_bankruptcy}\ \texttt{id}_x), \ldots], (\texttt{eligible\_for\_deferment}\ \texttt{id}_x) \rangle$$
$$y = \langle [(\texttt{disabled}\ \texttt{id}_y), \ldots], (\texttt{eligible\_for\_deferment}\ \texttt{id}_y) \rangle$$

Example x concerns a student that has filed for bankruptcy and according to the expert should be eligible for deferment, but when reasoning with the faulty rules the system solution will not match that of the expert's. Therefore, the KRUSTtool will attempt to refine the faulty rules by either generalising $R16$ or $R19$, by deleting condition (*enlist ?Student*), or (*disabled ?Student*), respectively. Let

us assume that the KRUSTtool chooses to refine by incorrectly generalising $R19$ (instead of $R16$) and implements this as a new KBS. On proceeding to the next refinement cycle (now with new KBS) the KRUSTtool is presented with fault evidence from training example y, a disabled student who is eligible for deferment. A direct consequence of generalising $R19$ is that the KRUSTtool is now left with no obvious refinement that can fix the fault exposed by y. Consequently, it is forced to re-think its previous refinement choice of generalising $R19$ instead of $R16$, and so faces the prospect of re-starting refinement from a previous state. Notice that if $R19$ and $R16$ were corrupted, but had no common condition that matched observables from either x or y (for instance like `filed_for_bankruptcy`) then the faults exposed by x and y in Figure 4 would not be interacting.

```
(defrule R16
   (filed_for_bankruptcy ?Student) (enlist ?Student)
   => (assert (financial_deferment ?Student)))

(defrule R19
   (disabled ?Student) (filed_for_bankruptcy ?Student)
   => (assert (disable_deferment ?Student)))

(defrule R10
   (financial_deferment ?Student)
   => (assert (eligible_for_deferment ?Student)))

(defrule R12
   (disable_deferment ?Student)
   => (assert (eligible_for_deferment ?Student)))
```

Fig. 4. Some rules taken from a corrupted student loans advisor in Clips.

The presence of interacting faults affects the refinement process, because selecting a non-optimal refined KBS in a previous iteration can cause refinement conflicts in a subsequent iteration. Detecting and resolving these refinement conflicts is important, as we have found that this improves refinement accuracy and guides the search for the best incremental refinements [13]. However, such conflicts can only be detected subject to the availability of fault evidence provided by a pair of examples, a *conflict pair* (such as x and y above). If a cluster contains *conflict pairs* like these, we would want to select further examples from this cluster. In these situations CLUSTERREP is not sufficient as it randomly selects a single example from each cluster, thereby ignoring all other examples in that cluster, including conflict pairs. A mechanism is needed to identify conflict pairs when they occur in the same cluster so that we ensure that examples exposing interacting faults are chosen. This necessitates an investigation of the problem-solving behaviour of labelled conflict pairs that occur in the same clus-

ter. The aim of such an investigation is to establish criteria that would enable the identification and selection of conflict pairs from a cluster when still unlabelled.

3.5 Characteristics of Conflict Pairs

An analysis of *labelled* conflict pairs revealed that they tend to have overlapping positive problem graphs, yet the best repair choices for the pair are distinguished from each other. Essentially their proofs may exercise similar parts of the KBS but their best repair exercises separate parts. Figure 5 shows the problem-solving for such a pair, C=$\langle [C_1, \ldots, C_6] \,|goal_C\rangle$ and D=$\langle [D_1, \ldots, D_6] \,|goal_D\rangle$. The darkened arrows and bold rule names highlight the positive problem graphs for examples C and D; i.e. the rules that are activated by the observables for each example. Each has resulted in the activation of the same end rule R_3, but the solutions (sys_C and sys_D) might occur with different variable bindings. Invariably a pair like this, with a substantial area of the positive problem graph in common, will be placed in the same cluster, and easily mistaken as representing the same fault.

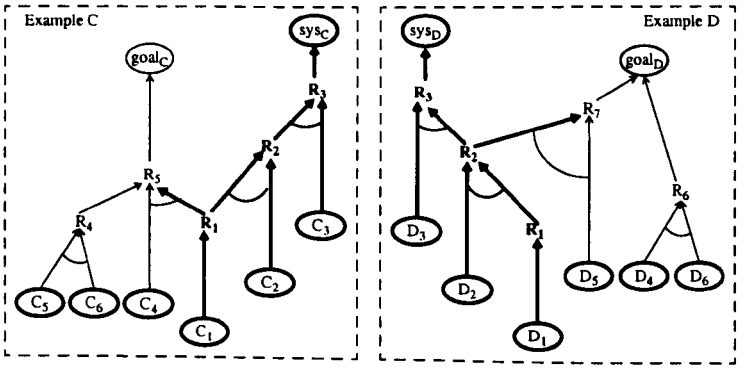


Fig. 5. Illustrating conflict pairs.

Figure 5 also shows all rules that might have concluded each target goal if they had been activated; these (non)activations form the *negative problem graph*. With example C, R_5 is only partially satisfied by R_1's conclusion. The arrow from C_4 is fainter to indicate that this condition in R_5 is not met by the observable without the condition being generalised somehow. The other possible route via R_4 requires both of its conditions to be generalised before being satisfied by C_5 and C_6. Possible repairs attempt to specialise rules in the positive problem graph and generalise those from the negative problem graph[1]. However,

[1] For a comprehensive list of KRUSTtool's specialisation and generalisation refinement operators see [14].

specialising R_2 to disallow the proof of sys_C for example C may cause problems when generalising R_7 to allow the proof of $goal_D$, for example D, and vice versa with R_1 and R_5. Essentially, even though conflict pairs are clustered together, a repair for one example will not necessarily repair the other; i.e. their negative problem graphs are fairly disjoint.

3.6 Informed Selection Heuristics

When examples are *unlabelled* we do not know the goals and cannot build the negative problem graphs. Instead we identify potential conflict pairs by formulating an indirect estimate of how overlapping the two negative problem graphs might be. For this purpose we compare their observables since the (non)activations in the negative problem graph depend on them.

We calculate a dissimilarity score for an example $e_i = \langle [f_1^i, \ldots, f_m^i], ? \rangle$, in a cluster $C = e_1, \ldots, e_n$ by summing all pair-wise dissimilarities between example e_i and the remaining examples in C.

$$Dissimilarity(e_i, C) = \sum_{j \neq i} dissimilarity(e_i, e_j)$$

$$dissimilarity(e_i, e_j) = \sqrt{\sum_{k=1}^{m} \delta^2(f_k^i, f_k^j)}$$

$$\delta(x, y) = \begin{cases} 0 & \text{if } x = y \\ \|n_x - n_y\| & \text{if } x, y \text{ are numeric facts}^2 \\ 1 & \text{otherwise} \end{cases}$$

The dissimilarity score of a cluster is the average *Dissimilarity* of its examples. There is some argument for ignoring the influence of observables that have already resulted in activations when calculating the dissimilarity score, however, as the contribution towards dissimilarity from observables associated with activations, compared to those associated with (non) activations is negligible, we have opted for the simpler *dissimilarity* score using all observables.

When a cluster has a high dissimilarity score there is reason to believe that such a cluster may contain conflict pairs, and we want to select it first for refinement. The intuition behind this is that examples clustered together based on similarity of the KBS's problem solving behaviour would normally also be similar in their observables. If observables are dissimilar then it is likely that problem solving behaviour of the KBS for that cluster is faulty and would require the selection of more than one example to fix the faults. We propose several sample selection heuristics that select varying numbers of examples from the cluster with the highest dissimilarity as follows: *DISSIMILAR selects *all* examples; K-DISSIMILAR, selects the K most dissimilar examples; and >DISSIMILAR selects examples with *Dissimilarity* scores above a pre-determined threshold.

[2] A numeric fact x has a numeric component n_x; e.g., age(fred, 40). $\|n_x - n_y\|$ is the absolute difference normalised by the range of values.

4 Experimental Evaluation

Example selection employing CLUSTERREP and the DISSIMILAR family of selection techniques are compared against RANDOM, where refinement examples are selected randomly. Our experiments test whether selective sampling produces refined KBSs with comparable accuracy but using fewer labelled examples than RANDOM. Furthermore, the performance of these techniques in the presence of interacting and non-interacting faults is also analysed by controlled corruptions of the KBS.

The data set and rule-base for the binary class student loans, and the data set for the multi class soybean was taken from the UCI repository [15]. The student loans data set consisted of 1000 labelled examples. We heavily corrupted the student loans KBS to encourage conflict pairs; by introducing 5 faults to the 20 rules. The soybean data set of 337 labelled examples was formed by merging the large and small soybean data sets and selecting those examples classified in the first 15 classes. A soybean KBS with 44 rules was created by incorporating rule chaining into the rule set generated by `c4.5rules` [16]. This KBS was then corrupted in 7 places, by adding and modifying antecedents in rules covering 4 of the 15 classes. Unlike the student loans corruptions, these faults did not interact, therefore examples from different classes have distinct problem graphs.

For each domain, a set of 100 training examples and a further 100 evaluation examples are randomly selected from the data set. The KRUSTtool is run with increasing subsets of the 100 training examples. Although all examples in the data set are labelled for experimentation purposes, these labels are ignored until examples are selected from the training set for the refinement task. Therefore, the labelling step in the *select-label-refine* iterative process is implicit, and the stop criterion is that the refined KBS has 100% accuracy on the training examples after the refinement step. We note that in practice this criterion is not available, as only selected training examples will be labelled, but that refinement is a continuous process constrained by expert availability. The impact of informed selection on *efficiency* is determined by the percentage of unused (unselected) examples in the training set. The impact on *effectiveness* is determined by the accuracy of the final KBS on the evaluation set. The graphs show results averaged over 10 runs for each training set size. Significance results are based on a 95% confidence level and apply the Kruskal Wallis [17] non-parametric test as some results are not normally distributed. The optimum cluster fusion threshold and the *Dissimilarity* threshold for >DISSIMILAR, with each test domain was ascertained by experimenting with varying thresholds, on a separate subset of examples.

4.1 Student Loans Domain

Experiments indicate that informed selection methods were effective: there was no significant difference in final refined KBS accuracy on the evaluation set, between these methods and RANDOM. Figure 6 shows the graph for unused percentage of examples for each of the methods. We found a significant difference

between these selection methods for unused percentage (p=0.005). 3-DISSIMILAR overall has faired best, and on average is three times more efficient than RANDOM or CLUSTERREP. 3-DISSIMILAR and >DISSIMILAR have significantly higher unused percentages compared to *DISSIMILAR, suggesting that the subset of most dissimilar examples from the cluster effectively targets the faults highlighted by all the examples in the cluster. All DISSIMILAR methods use significantly fewer training examples compared to CLUSTERREP and RANDOM. CLUSTERREP's poor performance is due to the added complication of interacting faults, and shows that selection of cluster representatives, alone, is not sufficient in these situations. The increase in unused percentage with training set size 10, seen with all methods, is explained by small training sets being insufficient to expose all faults in the KBS. As a result 100% accuracy on the training set is achieved easily, while the accuracy on the evaluation set will be significantly worse when compared to refined KBSs produced from larger training sets.

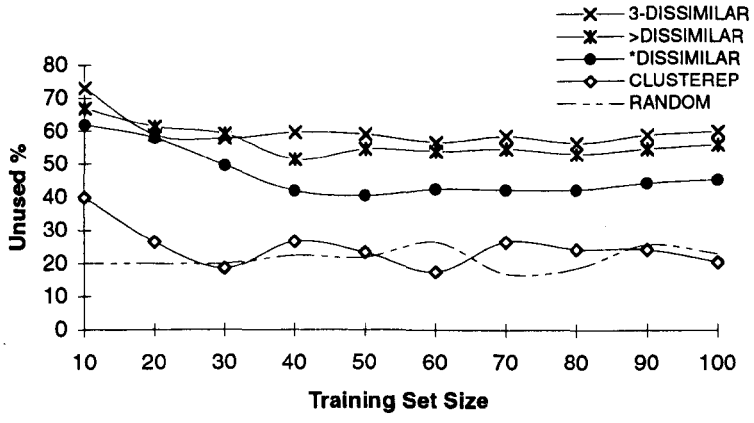


Fig. 6. Unused examples for student loans domain.

4.2 Soybean Disease Domain

Again there was no significant difference in accuracy between the selective methods and RANDOM; while there was a significant difference in unused percentages (p=0.005). From the efficiency view, in this domain, CLUSTERREP, uses significantly fewer examples than *DISSIMILAR and RANDOM (see Figure 7). The success of CLUSTERREP and the failure of *DISSIMILAR is explained by the absence of interacting faults in this rule base. Furthermore, the performance of CLUSTERREP improves with increased training set sizes, indicating that it was able to target few, yet good, examples. Closer examination of test runs with set sizes 70, 80, 90 and 100, revealed that the number of clusters tends to be constant while the size of clusters increases with the increasing number of examples,

therefore, CLUSTERREP selects the same number of examples regardless of the increase in set size. On average CLUSTERREP is three-times more efficient than RANDOM or *DISSIMILAR. *DISSIMILAR's bad performance with larger training set sizes clearly shows that the absence of an appropriate selection mechanism can result in ultimately using all the unlabelled examples. We have not plotted results for 3-DISSIMILAR and >DISSIMILAR methods as they are derivatives of *DISSIMILAR, which has performed poorly.

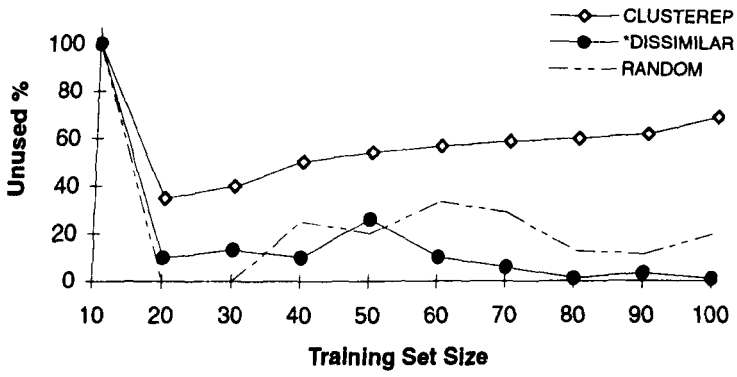


Fig. 7. Unused examples for soybean disease domain.

5 Related Work

The batch version of the refinement tool EITHER also applies incremental learning [18]. It processes batches of examples as they become available, but these examples are not selected for a purpose. Eventually EITHER uses all the examples, and in addition all these examples must be labelled. The use of membership queries and equivalence queries to select examples for learning Horn clauses is presented in [19]. Querying in this manner enables Horn clause learning in polynomial time. However, there is the assumption that labels of examples are known, and more importantly the logic based approach does not adapt well to rule-based systems that have more complex knowledge representation formalisms. EXPO [10] uses selective sampling to filter its proposed plans when the expected outcome of the plan differs from the actual observations. Interestingly EXPO's active selection occurs at plan filtering, analogous to the KRUSTtool's filtering of refined KBSs, and not for actively selecting planning tasks that may trigger learning, hence improving plan formation. This difference with knowledge refinement is possibly explained by the high costs associated with experimentation compared to access to representative planning problems.

Selective sampling employing a neural network for the task of learning a binary concept is discussed in [20]. An example is selected when the most specific

and most general network configurations fail to agree on the example's label. With complex concepts the most general network configuration may contain the entire domain, thereby forcing random sampling. Our clustering has similar problems: when the cluster threshold is too high, clusters contain single examples; when set too low one large cluster contains all examples. With each extreme selective sampling is reduced to RANDOM. Presently, we identify the optimum threshold by experimentation, however, the ability to automatically learn this threshold would be beneficial.

Argamon-Engelson and Dagan in [21] use a query by committee approach to selectively sample training examples for a probabilistic classifier. A committee of classifiers is randomly drawn based on statistics of the labelled sample. Examples are selected according to the degree of disagreement in class labels between the committee members. The committee approach can also be incorporated in knowledge refinement where the generated refined KBSs can vote on the solution for remaining training examples and select examples where the committee was unable to reach consensus. However, a disagreement measure is complicated when the KBS concludes in intermediate results.

Conceptual clustering involves arranging objects into clusters which would then represent certain conceptual classes [22]. However, such techniques require that there is some knowledge about the number of classes or, alternatively, knowledge about the goals of the classification. Usually, with knowledge refinement, there is no prior knowledge about the number of areas of the KBS that are faulty much less the types of faults that need to be addressed. However, our example clustering via rule vectors draws close parallels to classical document clustering in information retrieval where documents are represented as binary term vectors [23]. For information retrieval purposes documents with similar term vectors are grouped together forming a cluster. In document clustering, weights may also be used to indicate the relative importance of terms. We currently assign equal importance to all rule activations. However, a conservative view prefers refinements to rules closer to observables and this might be captured by introducing weights to rule activations.

6 Conclusion

We have presented an initial approach to selective sampling of training examples in the context of knowledge refinement. Experimental results show that selective sampling can significantly reduce the number of examples utilised, without any penalty on final accuracy. The refinement process was able to target particular faults that improved the accuracy of the refined KBS in a way that was effective in general. Not only did this reduce the number of refinement cycles required to achieve a particular level of competence, but it also reduced the demands on the expert's time. The selection was done based on features of the problem-solving task alone and so the expert was consulted about only the selected examples. Once labelled, the selected examples were presented to the refinement tool for processing.

The rule vector representation of the positive problem graph provided a simple similarity measure that created clusters of examples that had been solved by the KBS in a similar way. This clustering was helpful in determining examples that might indicate the same repair. Future work will analyse the implications of rule depth and the sequence of rule activations on similarity and investigate how the similarity measure might be extended to reflect these. Given a clustering, incremental refinement can be visualised by capturing changes in cluster size and cluster membership. We are currently exploiting these dynamic changes for example selection during the refinement filtering stage, where the aim is to identify examples affected by the proposed refinements. We note that this is possible due to our clustering using similarity between, rule vectors rather than feature vectors, as employed by most existing active learning methods.

The difficulty of selecting examples from clusters depends on the level of interaction of the faults in the KBS. Experiments have highlighted the strengths of DISSIMILAR heuristics in the presence of interacting faults and the less informed CLUSTERREP selection heuristic in the presence of non interacting faults. We intend to develop more powerful selection mechanisms that combine these techniques. One possibility would be to choose between selection heuristics CLUSTERREP and a DISSIMILAR method after a clustering has been done: if the maximum intra cluster dissimilarity is large then a DISSIMILAR method is required; if small then CLUSTERREP is sufficient.

Selective sampling is important for knowledge refinement tools whether or not labelled training examples are plentiful. If labels are hard to obtain then it is certainly useful to identify relevant problem-solving tasks that should be labelled by the expert and then used as training examples for refinement. Conversely if there are many labelled training examples then, given that the refinement process is quite computationally expensive, it is convenient to target those examples whose repairs also fix other wrongly solved examples without further refinement, thereby reducing the number of refinement cycles. Selective sampling addresses both these issues by identifying the examples most likely to solve others that indicate the same general fault.

Acknowledgments

The KRUSTWORKS project is supported by EPSRC grant GR/L38387 awarded to Susan Craw. Nirmalie Wiratunga is partially funded by ORS grant 98131005.

References

1. Susan Craw and Robin Boswell. Representing problem-solving for knowledge refinement. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pages 227–234, Menlo Park, California, 1999. AAAI Press.
2. Marcelo Tallis and Yolanda Gil. Designing scripts to guide users in modifying knowledge based systems. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pages 227–234, Menlo Park, California, 1999. AAAI Press.

3. B. Richards and R. Mooney. Automated refinement of first-order horn-clause domain theories. *Machine Learning*, 19:95–131, 1995.
4. N Zlatareva and A Preece. State of the art in automated validation of knowledge-based systems. *Expert Systems with Applications*, 7:151–167, 1994.
5. Dmitry Rusakov Michael Lindenbaum, Shaul Markovich. Selective sampling for nearest neighbor classifiers. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pages 366–371, Menlo Park, California, 1999. AAAI Press.
6. Andrew McCallum and Kamal Nigam. Employing em in pool-based active learning for text classification. In *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 359–367, 1998.
7. David D. Lewis and Jason Catlett. Heterogeneous uncertainty sampling for supervised learning. In William W. Cohen and Haym Hirsh, editors, *Machine Learning: Proceedings of the Eleventh International Conference*, pages 148–156, San Francisco, CA, 1989. Morgan Kauffman.
8. David Cohn, Zoubin Ghahramani, and Michael I. Jordan. Active learning with statistical models. *Journal of Artificial Intelligence Research*, 4:129–145, 1996.
9. Robin Boswell, Susan Craw, and Ray Rowe. Knowledge refinement for a design system. In *Proceedings of the Tenth European Knowledge Acquisition Workshop*, pages 49–64, Sant Feliu de Guixols, Spain, 1997. Springer.
10. Yolanda Gil. Learning from the environment by experimentation: The need for few and informative examples. In *Proceedings of the AAAI Symposium on Active Learning*, MIT, Cambridge, MA, 1995.
11. Peter Willett. Recent trends in hierarchic document clustering: A critical review. *Information Processing and Management*, 24:577–597, 1988.
12. Stephen J. Hanson. Conceptual clustering and categorization. In Y. Kodratoff and R. S. Michalski, editors, *Machine Learning Volume III*, pages 235–268. Morgan Kaufmann, San Mateo, CA, 1990.
13. Nirmalie Wiratunga and Susan Craw. Sequencing training examples for iterative knowledge refinement. In *Proceedings of the Nineteenth SGES International Conference on Knowledge Based Systems and Applied Artificial Intelligence*, pages 41–56, Cambridge, UK, 1999. Springer.
14. Robin Boswell and Susan Craw. Organising Knowledge Refinement Operators In *Validation and Verification of Knowledge Based Systems, Proceedings of the 5th European Symposium on the Validation and Verification of Knowledge Based Systems (EUROVAV'99)*, pages 149–161, Oslo, Norway, 1999. Kluwer.
15. C. Blake, E. Keogh, and C.J. Merz. UCI repository of machine learning databases. http://www.ics.uci.edu/~mlearn/MLRepository.html, 1998.
16. J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, 1993.
17. D. A. Anderson, D. J. Sweeney, and T. A. Williams. *Statistics for Business and Economics*. West Publishing Company, St. Paul, MN, 1990.
18. Raymond J. Mooney. Batch versus incremental theory refinement. In *Proceedings of the AAAI Spring Symposium on Knowledge Assimilation*, Stanford, CA, 1992.
19. Dana Angluin, Michael Frazier, and Leonard Pitt. Learning conjunctions of horn clauses. *Machine Learning*, 9:147–164, 1992.
20. David Cohn, Les Atlas, and Richard Ladner. Improving generalization with active learning. *Machine Learning*, 15:201–221, 1994.
21. Shlomo Argamon-Engelson and Ido Dagan. Committee-based sample selection for probabilistic classifiers. *Journal of Artificial Intelligence Research*, 11:335–360, 1999.

22. R.S. Michalski and R.E. Stepp. Clustering. In S.C. Shapiro, editor, *Encyclopaedia of Artificial Intelligence*, volume 1, pages 103–110. Wiley, 1990.
23. Edie Rasumssen. Clustering algorithms. In W. B. Frakes and R. Baeza-Yates, editors, *Information Retrieval: Data Structures and Algorithms*, pages 419–442. Prentice Hall, London, 1992.