An Approach to User-Directed Search in Interactive Problem Solving

A thesis submitted for the degree of Master of Science of The Australian National University

Qin Yang

April 1991
An Approach to User-Directed Search
in Interactive Problem Solving
Acknowledgements

It has been a most rewarding experience for me to study in the Department of Computer Science. This is not only because of the academic abilities of the staff from whom I have learned so much, but also because of an atmosphere of friendship and close co-operation throughout the department fostered by the Head, Robin B. Stanton, which has made the learning very enjoyable.

I am greatly indebted to my supervisor, Prof. Robin B. Stanton, for his constant support, cheerful encouragement and expert advice in all stages of my research. His scientific insight and patient guidance have led me into the interesting field of expert system research and have enabled this research to be accomplished.

I would like to express my gratitude to Dr. Vicki Peterson and Dr. Qing Li for their helpful comments and suggestions which made the thesis more accurate and precise. Thanks are due to the programmers, David Hawking, Drew Corrigan and Ivan Dean, for their readily available technical assistance.

I should thank my fellow students: Graham Williams, Jason Lan, Seppo Keronen, Dan Zhou, Jian Yang, Jimmy Wang, Bill Zhou and Ely Albacea, who have provided so many friendly talks and useful discussions. In particular, I am very thankful to Dharmendra Sharma and James Popple for their kind help whenever it was requested.

I am very grateful to my close friend, Dr. Gang-Ding Peng, for his constant help and critical advice. I owe Dr. Adrian Ankiewicz and Stan Jones many thanks for their friendship and proof-reading of the manuscript.

I wish to thank my parents for their love, encouragement and support.

Finally, my thanks go to the Australian National University for much appreciated
Acknowledgements

financial support.
Declaration

This thesis describes the original work of the author. Where this work is based on the work of others, this is clearly indicated in the text.

Qin Yang
Abstract

This thesis studies some problems which are important in establishing interactive problem solving systems. An interactive problem solving system is characterized by the intensive interaction between the user and the system. In order to converge on a solution which satisfies the user, we present a new problem solving scheme – user-directed search (UDS) – where the solution search is directed in a step-by-step manner by the user. Because of its wide applicability, UDS can be very useful for many practical cases.

The user-directed problem solving is realized by introducing a particular communication mechanism between the user and the system. This enables a user to guide the solution searching in his most preferred directions. Thus the system can first explore the solutions which are more likely to match the user-desired solution.

We have developed UDS using two different approaches.

In the first approach, additional deduction rules can be created upon the user’s request and/or upon changes in practical environments. For this purpose, we have created, in the user interface, an environment which enables a user to add his new requirements in the form of deduction rules. To improve efficiency, we have used a particular backjump search which can first find, and then backjump to, the point which contradicts the user’s new requirements. To establish the dependency for this backjumping, we have used assumption-based truth maintenance systems (ATMS) and KEEworlds in the knowledge engineering environment(KEE).

In the second approach, we have introduced particular variable groups. In this approach, the user’s new requirements are introduced through a scheme in which the user divides the variable set into several different variable groups. By dividing these
variable groups according to his choice, a user can effectively control and instruct the search during the process of problem solving. We have introduced here a scheme which we call proximal minimum (closeness) change. The proximal minimum change ensures that, in the direction specified by the user, a closest solution to the previous one will be found if it actually exists.

In another aspect, in order to improve efficiency of solution search on a general basis, we have applied some techniques from Constraint Satisfaction Problems (CSP) in establishing non-CSP expert systems, e.g. rule-based and frame-structured expert systems on KEE. We find that these CSP techniques can be used to improve efficiency by performing consistency checking prior to searching for a solution, which we call pre-processing. This pre-processing is introduced to eliminate a number of variable values which are inconsistent with certain unary and binary constraints. In practical applications, this method can be used to avoid a considerable amount of useless backtracking. We have developed an independent module for applying CSP techniques in general purpose programming in KEE. This CSP module provides KEE with ability to establish more versatile expert systems.

Through case studies of the truck dispatching problem and the word puzzle problem, we demonstrate how to achieve UDS and how to implement various techniques which we have presented to improve efficiency in UDS. Some of the advantages of UDS are shown in the case studies.
# Contents

1 Introduction .................................................. 1

1.1 Expert Systems ........................................... 1

1.2 Problem Solving ........................................... 5

1.3 Search Strategies ........................................ 7

1.4 Design Goals ............................................... 8

1.5 Overview of Thesis ...................................... 11

2 KEE, ATMS, KEEworlds ...................................... 15

2.1 KEE ........................................................ 16

2.1.1 Knowledge Representation .............................. 17

2.2 Assumption-Based Truth Maintenance System (ATMS) .... 21

2.2.1 Deduction Rules and Justifications ................... 22

2.2.2 Constraints ............................................ 22

2.3 KEEworlds .................................................. 23

2.3.1 Worlds ................................................. 24

2.3.2 Worlds and Forwarding Chaining ..................... 26

2.3.3 Worlds and Justifications ............................. 26

3 Constraint Satisfaction Problem Techniques ................. 29

3.1 Introduction ................................................. 29

3.2 Constraint Satisfaction Problems Definition ............... 30

3.3 A Network Representation of a CSP ....................... 34

3.4 Inconsistency and Consistency ........................... 35

3.4.1 Node Consistency Checking ........................... 37
3.4.2 Arc Consistency Checking ........................................ 38
3.4.3 Path Consistency Checking ....................................... 39
3.5 Summary ............................................................... 39

4 Implementation of CSP Techniques in KEE 41
4.1 Introduction .......................................................... 41
4.2 CSP Representation in KEE ......................................... 42
   4.2.1 NODES unit ....................................................... 44
   4.2.2 ARCS Unit ...................................................... 50
   4.2.3 PATHS unit ..................................................... 54
   4.2.4 BOSS unit ....................................................... 54
4.3 Consistency Checking Implementation in KEE ..................... 57
   4.3.1 NODE.CONSISTENCIES Unit ................................ 57
   4.3.2 ARC.CONSISTENCIES Unit .................................. 58
   4.3.3 PATH.CONSISTENCIES Unit ................................. 62
   4.3.4 An Example ..................................................... 63
4.4 Discussion ............................................................ 66

5 Two Strategies for User-Directed Search 67
5.1 Introduction .......................................................... 67
5.2 UDS ................................................................. 68
5.3 Backjumping .......................................................... 69
5.4 Constraint Recording ................................................. 75
5.5 Variable Classification .............................................. 76
5.6 Comparison of two strategies ..................................... 80

6 Case Study I: Truck Dispatching Problem 83
6.1 Introduction .......................................................... 83
6.2 Problem Description ................................................ 85
   6.2.1 Static Constraints ............................................. 85
   6.2.2 Dynamic Constraints ......................................... 86
6.3 System Overview ..................................................... 87
Contents

6.3.1 Flow of Control ........................................... 87
6.3.2 Design Features ........................................... 88
6.4 Knowledge Representation ................................. 90
   6.4.1 Truck Dispatching Knowledge ......................... 90
   6.4.2 Constraint Rules and Dispatching Rules ............ 94
6.5 CSP for Data Pre-Processing .............................. 97
6.6 Constraint-Based Search .................................. 99
   6.6.1 Constraints and Search ............................. 100
   6.6.2 Rule-Based Reasoning .............................. 101
6.7 Constraint Recording While Searching ................... 105
6.8 Backjumping for UDS .................................. 107
6.9 Summary .................................................. 110

7 Case Study II: Crossword Puzzle Problem ................. 111
   7.1 Introduction .......................................... 111
   7.2 Problem Description and Knowledge Representation ... 113
   7.3 Searching Scheme ...................................... 115
      7.3.1 Deduction Rules for Constraints ................ 115
      7.3.2 Action Rules for Search ......................... 117
   7.4 User-Directed Search Implementation .................. 121
      7.4.1 Variable Classifying ............................ 121
      7.4.2 Domain Reordering .............................. 123
      7.4.3 Variable Reordering ............................ 125
      7.4.4 Minimum Changes .............................. 126
   7.5 CSP Techniques for Filtering .......................... 128
      7.5.1 Data Pre-Processing ............................ 128
      7.5.2 Two Level Filters in UDS ....................... 129
   7.6 An Example of UDS .................................... 131
   7.7 Summary ............................................... 133

8 Conclusions and Further Directions ........................ 135
## Contents

### Appendix

- 139

### Bibliography

- 145
List of Figures

1.1 A schematic expert system ........................................ 2
2.1 The main packages and facilities in KEE. ...................... 16
2.2 *in.new.world* creates worlds under an exclusion set. ........ 20
2.3 A world graph from truck dispatching problem. ............... 25
3.1 An example backtrack search tree. ............................... 34
3.2 The constraint network of a graph-colouring problem. ......... 35
4.1 The representation of NODES unit and its descendants ........ 45
4.2 ARCS unit and its descendants. .................................. 51
4.3 PATHS unit and its descendants. ................................ 53
4.4 NAP CONSISTENCIES unit and its descendants ................. 57
4.5 A constraint satisfaction problem: crossword puzzle. ......... 63
4.6 (a). The original constraint network for the crossword puzzle. (b). The constraint network after applying the NAPCC module for node consistency checking. ........................................ 64
4.7 (a). The detailed process of performing arc consistency checking. (b). The constraint network after applying the NAPCC module for arc consistency checking. ........................................ 64
4.8 A solution for a crossword puzzle. ............................... 65
5.1 The algorithm of searching for a possible solution. .......... 71
5.2 Main control algorithm and UDS1. ............................... 73
5.3 An algorithm for collecting user's new requirements. ......... 74
List of Figures

5.4 Backjumping algorithm of UDS1. ........................................... 74
5.5 Main control Algorithm for UDS2. ........................................ 77
5.6 The algorithm of searching for a possible solution for UDS2. ......... 79
5.7 Algorithm for UDS2. ............................................................... 81
5.8 The algorithm of closeness change on the variables in the base group. 82

6.1 The control structure of a truck dispatching system. .................... 89
6.2 Trucks hierarchy for representing all the trucks. .......................... 91
6.3 The constraint network of a truck dispatching problem. ................. 98
6.4 Search spaces, candidate trucks and drivers before and after perform-
ing data pre-processing. .............................................................. 99
6.5 The rule hierarchy used for our truck dispatching problem. ............. 103
6.6 An example of the information accumulated during searching. ........ 106
6.7 An example of backjumping in UDS. ......................................... 109

7.1 A crossword puzzle as a constraint satisfaction problem. ............... 114
7.2 The constraint network for the crossword puzzle in Figure 7.1. ......... 114
7.3 The process of searching for a possible solution. .......................... 119
7.4 Rule hierarchy organization for solving a crossword puzzle problem. 120
7.5 Domain reordering during processing. ....................................... 124
7.6 The cases where only one variable is revised. ............................. 126
7.7 The cases where variable $v_b_i$ and any one other variable are revised. 127
7.8 Case where all the variable are changed. .................................... 127
7.9 The search tree for the base group. ......................................... 128
7.10 A possible solution of the crossword puzzle in Figure 7.1. ............. 130
7.11 A search space within user-directed search. ............................... 131
List of Tables

4.1 NODES unit and its internal structure. ........................................ 46
4.2 NODE1 unit and its internal structure. ...................................... 48
4.3 ARCS unit and one of its descendant ARC1 and part of their internal structure. .................................................. 51
4.4 PATHS unit with one of its descendants, and part of their internal structure. .................................................. 53
4.5 BOSS unit and part of its internal structure. .............................. 55
4.6 NODE.CONSISTENCIES unit and its internal structure. .............. 58
4.7 NODE.CONSISTENCY unit and its internal structure. ................... 59
4.8 ARC.CONSISTENCIES unit and its internal structure. .................. 59

6.1 The attributes of Trucks. ...................................................... 92
6.2 Details of the truck T4. ....................................................... 93
6.3 The attributes of Drivers. ..................................................... 93
6.4 The Relations of Trips. ........................................................ 94
List of Tables

1. The analysis structure of a track positioning system

2. The analysis structure of a track positioning system

3. The analysis structure of a track positioning system

4. The analysis structure of a track positioning system

5. The analysis structure of a track positioning system

6. The analysis structure of a track positioning system

7. The analysis structure of a track positioning system

8. The analysis structure of a track positioning system

9. The analysis structure of a track positioning system

10. The analysis structure of a track positioning system

11. The analysis structure of a track positioning system

12. The analysis structure of a track positioning system

13. The analysis structure of a track positioning system

14. The analysis structure of a track positioning system

15. The analysis structure of a track positioning system

16. The analysis structure of a track positioning system

17. The analysis structure of a track positioning system

18. The analysis structure of a track positioning system

19. The analysis structure of a track positioning system

20. The analysis structure of a track positioning system

21. The analysis structure of a track positioning system

22. The analysis structure of a track positioning system

23. The analysis structure of a track positioning system

24. The analysis structure of a track positioning system

25. The analysis structure of a track positioning system

26. The analysis structure of a track positioning system

27. The analysis structure of a track positioning system

28. The analysis structure of a track positioning system

29. The analysis structure of a track positioning system

30. The analysis structure of a track positioning system

31. The analysis structure of a track positioning system

32. The analysis structure of a track positioning system

33. The analysis structure of a track positioning system

34. The analysis structure of a track positioning system

35. The analysis structure of a track positioning system

36. The analysis structure of a track positioning system

37. The analysis structure of a track positioning system

38. The analysis structure of a track positioning system

39. The analysis structure of a track positioning system

40. The analysis structure of a track positioning system

41. The analysis structure of a track positioning system

42. The analysis structure of a track positioning system

43. The analysis structure of a track positioning system

44. The analysis structure of a track positioning system

45. The analysis structure of a track positioning system

46. The analysis structure of a track positioning system

47. The analysis structure of a track positioning system

48. The analysis structure of a track positioning system

49. The analysis structure of a track positioning system

50. The analysis structure of a track positioning system

51. The analysis structure of a track positioning system

52. The analysis structure of a track positioning system

53. The analysis structure of a track positioning system

54. The analysis structure of a track positioning system

55. The analysis structure of a track positioning system

56. The analysis structure of a track positioning system

57. The analysis structure of a track positioning system

58. The analysis structure of a track positioning system

59. The analysis structure of a track positioning system

60. The analysis structure of a track positioning system

61. The analysis structure of a track positioning system

62. The analysis structure of a track positioning system

63. The analysis structure of a track positioning system

64. The analysis structure of a track positioning system

65. The analysis structure of a track positioning system

66. The analysis structure of a track positioning system

67. The analysis structure of a track positioning system

68. The analysis structure of a track positioning system

69. The analysis structure of a track positioning system

70. The analysis structure of a track positioning system

71. The analysis structure of a track positioning system

72. The analysis structure of a track positioning system

73. The analysis structure of a track positioning system

74. The analysis structure of a track positioning system

75. The analysis structure of a track positioning system

76. The analysis structure of a track positioning system

77. The analysis structure of a track positioning system

78. The analysis structure of a track positioning system

79. The analysis structure of a track positioning system

80. The analysis structure of a track positioning system

81. The analysis structure of a track positioning system

82. The analysis structure of a track positioning system

83. The analysis structure of a track positioning system

84. The analysis structure of a track positioning system

85. The analysis structure of a track positioning system

86. The analysis structure of a track positioning system

87. The analysis structure of a track positioning system

88. The analysis structure of a track positioning system

89. The analysis structure of a track positioning system

90. The analysis structure of a track positioning system

91. The analysis structure of a track positioning system

92. The analysis structure of a track positioning system

93. The analysis structure of a track positioning system

94. The analysis structure of a track positioning system

95. The analysis structure of a track positioning system

96. The analysis structure of a track positioning system

97. The analysis structure of a track positioning system

98. The analysis structure of a track positioning system

99. The analysis structure of a track positioning system

100. The analysis structure of a track positioning system

101. The analysis structure of a track positioning system

102. The analysis structure of a track positioning system

103. The analysis structure of a track positioning system

104. The analysis structure of a track positioning system

105. The analysis structure of a track positioning system

106. The analysis structure of a track positioning system

107. The analysis structure of a track positioning system

108. The analysis structure of a track positioning system

109. The analysis structure of a track positioning system

110. The analysis structure of a track positioning system

111. The analysis structure of a track positioning system

112. The analysis structure of a track positioning system

113. The analysis structure of a track positioning system

114. The analysis structure of a track positioning system

115. The analysis structure of a track positioning system

116. The analysis structure of a track positioning system

117. The analysis structure of a track positioning system

118. The analysis structure of a track positioning system

119. The analysis structure of a track positioning system

120. The analysis structure of a track positioning system

121. The analysis structure of a track positioning system

122. The analysis structure of a track positioning system

123. The analysis structure of a track positioning system

124. The analysis structure of a track positioning system

125. The analysis structure of a track positioning system

126. The analysis structure of a track positioning system

127. The analysis structure of a track positioning system

128. The analysis structure of a track positioning system

129. The analysis structure of a track positioning system

130. The analysis structure of a track positioning system

131. The analysis structure of a track positioning system
Chapter 1

Introduction

1.1 Expert Systems

Expert systems are artificial intelligence programs which are designed to achieve, to a certain extent, human expertise in some specific fields such as industrial design and manufacture, official administration, financial management, medical diagnosis and so forth. An expert system can be advantageous in various aspects: it can make expertise easy to access, store and apply; make problem-solving efficient and effective; and so on. Principally, an expert system may be designed to employ human expertise in certain related areas of a particular field. However, the demonstrated expert systems are mainly designed for a specialized area. Many such systems have been developed since the beginning of 1970s [Waterman 86], [Shapiro 87], [Grimson 87], [Brachman 90] and [Lenat 90].

Human intelligence is essentially characterized by knowledge and the ability to apply that knowledge to solve many practical problems. Knowledge is based on both the facts we have observed and experienced and the relations among them we have found. The ability to utilize knowledge allows us to derive new information from the knowledge already obtained. To build an expert system is to make an electronic system to store, apply, and even accumulate, knowledge in a way resembling a human being. Hence an expert system must incorporate a knowledge base (in Figure 1.1) to allow human knowledge to be represented, and a knowledge-applying...
process (inferencing engine in Figure 1.1) to simulate human thinking. As shown in
Figure 1.1, there are two other basic components in an expert system: a data base
and a user interface. The data base maintains all the information of operating an
expert system: initial data, intermediate data, system parameters and final results.
The user interface provides the necessary communication between the system and
the operator. It allows the operator to initiate a problem, to make an inquiry during
the operation of the system or to respond to a solution found.

A significant step in building an expert system is to construct a knowledge base.
There are quite a number of methods that have been developed for expressing spe­
cialized knowledge in expert systems. These include methods involving production
rules, constraints, frames, semantics nets, and so forth. The most extensively used
method for knowledge representation is ‘production rules’, because it is very modu­
lar and flexible. Most rules have the form: ‘an \textbf{if} \cdots \textbf{then} \cdots \textbf{clause}’ which consists
of two parts: the first part states premises or conditions and the second part states
conclusions or actions. Expert systems with the knowledge formatted into a set of

Figure 1.1: A schematic expert system
1.1. Expert Systems

rules are called rule-based systems.

Here is an example rule from a truck dispatching problem [Filman 88]:

if: there is a trip to be assigned a driver; and
there is a candidate driver;
then:
assign the driver to the trip;
invoke constraint rules;

Another important and powerful knowledge representation method is to use constraints. With this method, knowledge, that is, facts and the relations among them, is expressed as a set of constraints. An expert system problem with the knowledge formatted into a set of constraints is called a constraint satisfaction problem.

In the truck dispatching problem, examples of constraints include “You cannot put more on a truck than it can hold” and “A driver needs to have the right licence to drive a particular truck”. An expert system incorporates constraints to express restrictions on allowable states, values, or conclusions. In fact, some systems derive their solutions by analysing and satisfying complicated constraints. Generally, these programs exploit expressions with variables to denote class of objects or relationships among classes of objects. In addition, they incorporate additional requirements that constrain the possible value assignments which variables can take. They use inference rules to determine which variable values can be derived from others.

We can make good use of constraint sets to perform consistency checking (a detailed description can be found in Chapter 3 and 4) before searching for a possible solution, so that the search space can be reduced. The development and research in this area has led up to the current great interest in expert systems.

Different knowledge representation methods have quite different properties which might be advantageous in one case but disadvantageous in another. Thus it is important to carefully select a suitable method for a particular expert system application. Moreover, for some expert systems, part of the knowledge may be best expressed in one format while other parts are best expressed in others. In particular, in our truck dispatching system (to be studied in Chapter 6), we have undertaken to improve
searching efficiency by establishing a knowledge base with both production rules and constraint rules.

The inference engine is the heart of an expert system. It is the central software which controls and coordinates the knowledge base, data base and user interface and which employs specially selected searching strategies to find solutions to a given problem. The inference engine works in a way that, after initial 'ignition' – it starts and will run automatically to find the solution to any problem given by the user. By using the formatted knowledge, the engine goes through, in a well-controlled sequence, the rules and constraints stored in the knowledge base and examines this knowledge with the information given in the data base. The examining process in rule-based systems is typically pattern-matching. The process in a constraint satisfaction problem is usually called consistency checking.

The data base is an important part of an expert system because it can maintain and manage various data such as initial facts of the problem under consideration and intermediate facts created on the track of inferencing operations. User interface provides a facility that allows the user to initiate an inquiry, to monitor or guide the inferencing process and that, in some cases, allows the system to ask the user about information when it is considered necessary.

To make it easier to establish expert systems, great efforts have been made to construct expert system tools. An expert system tool can be considered as a designated programming environment which is built on some universal aspects of an expert system design. In contrast to the usual programming languages, expert system tools take facts, rules as their objects and pattern-matching or inference engine as their control structure. A large number of expert system tools from personal computers to mainframe computers have been developed in recent years [Benchimol 87]. For example, expert system development tools such as KEE[Intelicorp 87], Smalltalk [Goldberg 83], OPS-5 [Brownston 85] and ART [Williams 84] have been made commercially available. The case studies presented in this thesis (see Chapters 6 and 7) are carried out using KEE.
1.2 Problem Solving

Problem Solving is the central issue in artificial intelligence [Shapiro 87] and [Frenzel 87]. It is a process that involves finding, or constructing, a solution to a problem. The purpose of establishing an expert system is to solve problems requiring a certain kind of expertise. In general, problem solving involves every aspect of an expert system, because the elements of an expert system: knowledge base, inferencing engine, user interface, and so on, are highly interrelated. Usually, however, problem solving is referred to as a specific process which is guided by a control strategy to search for a feasible solution upon given conditions.

Formally, the process of solving a problem can be defined in the following general form [Shapiro 87]:

Given a domain specification $D$, find a solution $s$ such that $s$ is a member of a set of possible solutions $S$ and it satisfies the conditions $C$.

It is very important to effectively and efficiently exploit the information in $D$, $C$. The conditions $C$ specify the constraints to the problem under consideration. They can be applied to solution construction in two ways: a posteriori testing which examines candidate solutions and a priori constraining which constrains the generator (and possibly influencing the controller). Thus those candidate solutions which do not satisfy the constraints will be excluded until an acceptable solution is found.

Generally, a problem-solving system is designed to search for solutions to a given problem. We may use a variety of search techniques, and may process a plausible solution generator that can enumerate all possible solutions. Then the systems can generate and test all possibilities until an acceptable solution is found. The set of all the possibilities is called the search space and this approach to search is called brute force. Many ordinary problems, such as map coloring, database retrieval for conjunctive queries, Boolean satisfiability and so forth may have search space of astronomical size. Hence specific tools, methods and mechanisms must be introduced to effectively reduce search space to a tolerable dimension. The efficiency of a problem-solver may be improved by a strategy in which some domain values
that cannot succeed in the testing are excluded before searching begins.

Problem solving of constraint satisfaction problems (CSP) can be a good example. As mentioned above, many expert system tasks can be represented by constraint satisfaction problems. Some special techniques introduced to solve CSP are based on eliminating candidate solutions through successively refining value sets of the variables, that is, on deleting unacceptable values from the domains of all the variables. The process is carried out by checking the specific constraints so as to eliminate subsets of the set of candidate solutions that are inconsistent with the constraints. Following these candidate-elimination steps, the problem solver can proceed to find a solution using a generate-and-test scheme or one of its variants.

There may exist a number of solutions (the solution space) satisfying given constraint sets. In most cases, only one solution, which may not be the best solution, is enough for a given problem. In design of an expert system, computer time efficiency (time required to find a feasible solution) and space efficiency (memory required to store all the relevant information) are of prime concern. Hence, in these applications, a reasonable solution is accepted by the user, considering that far more effort might have to be spent in finding the best solution than would be on a reasonable solution.

There are many cases where a user may not be satisfied with the solution obtained to an initial problem for various practical reasons. For example, this situation may happen in some expert systems work in dynamically changing environments and where certain changes may occur in the facts or conditions during the problem solving. A number of questions could be raised in such cases. Is it necessary to start all over again to search for another solution? Is it possible to make use of the solution obtained, or, to inherit some information collected during the search of it?

Obviously, it is inefficient to abandon all the information collected and to start the search from the very beginning. However, knowing how to effectively make use of the information obtained is not simple because we know that even minor changes to the facts and/or initial conditions can lead to totally different solutions. Therefore, it is very important to investigate how to deal with this kind of problem in order to
One of our major concerns in this thesis is a focus on some fundamental problems in interactive problem solving. We will establish a mechanism which enables a user to guide and direct the solution searching in the most preferred directions and thus the user-desired solution can be obtained to a problem in the most efficient way.

1.3 Search Strategies

In an expert system, to solve a problem means either to answer a question or to make a decision based upon certain initial facts. We know that the most important step of problem solving in an expert system is the search [Shapiro 87] and [Pearl 84]. That is, search is a general problem-solving mechanism in expert systems. When a knowledge base has been established, search procedures can be implemented. In a search process, a certain kind of inferencing mechanism is introduced to reference the knowledge base in a certain orderly way to find the solution. The process of search is programmed in such a way that it usually examines the facts already in the knowledge base and collects, at the same time, new facts derived from the search itself in order to arrive at solutions to a given problem.

There are two basic ways of search: blind search and heuristic search [Frenzel 87] and [Shapiro 87].

Blind search explores solutions in a way which checks all the possibilities given by the relevant knowledge as well as the initial conditions. Blind search is inefficient because it takes considerable amount of time to find a solution. Especially for large systems, which contain many facts and constraints to be examined, blind search may lead to combinational explosion. In these cases, blind search seems quite impractical. However, blind search is useful for small expert systems where computing time will not be a problem considering of the small number of possibilities. In fact, blind search will usually produce a conclusion and, moreover, it always plays an important role in heuristic search where local blind search is used frequently.

Heuristic search is a solution-searching approach which uses various assistant means such as hints and rules of thumb to actively select certain privileged search
directions which, heuristically, are most likely to lead to a possible solution.

In practical cases, that is, the cases we will discuss, solution search uses a combination of both blind search and heuristic search in order to achieve proper efficiency while a reasonable number of facts and constraints are taken into consideration.

1.4 Design Goals

The efficiency of problem solving in expert systems is determined by many different factors. Among them, search strategy is an essential factor. However, even using the same search strategy, the efficiency may differ from case to case. Hence there seems no universal search strategy suitable for all the cases. Then how may an intelligent problem solver expect improved efficiency? Here we present some criteria which are important for a problem-solver to be efficient.

(1) Find and eliminate those paths of search which will finally lead to dead end.

Hence, classes of possibilities which will consume useful computing time can be removed from further consideration.

(2) Record and utilize the reason for a dead-end search so that the same conflict will not again in the later search. Redundancy in repetitively searching similar fruitless paths may thus be avoided.

(3) Make use of the results obtained to guide further solution searching.

In this study, we develop an interactive problem solving environment in which the solution search is accomplished by an interaction between the user and the system. In this regard, we introduce a new scheme which is called a User-Directed Search (UDS). This means that the user is allowed to guide and to control the solution search in the most preferred directions.

Interactive problem solving can provide a very useful scheme on which the solution search can be conveniently controlled, interrupted and directed. A new solution searching should always be based on the solution obtained plus the addition of further conditions. Thus we have studied how to achieve efficiency by making as much
1.4. Design Goals

use as possible of the solution obtained in further search. Taking this into consideration, we will present here a new method of interactive problem solving.

In this thesis, we have made considerable efforts to achieve UDS in solution search in the following directions:

1. Closeness change in UDS

In many expert systems, there may exist a number of solutions scattered in the solution space which are determined by the conditions corresponding to a given problem. It can be expected that the particular solution which is explored, and the order in which the solutions are explored will be determined by the search route being taken. It is common that the user may show preference in accepting one solution while declining another. For example, a solution found by the system may be sent back to the user interface and the user may find it unacceptable, because the solution is structured in a way differing from what he had expected or it conflicts with some conditions which had not been considered beforehand. In cases like this, how can a user find quickly a solution which he is ready to accept?

To solve such a problem we will introduce a user-directed mechanism in solution search. In particular, a good interface has to be established such that solutions in the solution space will be explored along the user’s preferred directions with proximal minimum (closeness) changes from one solution to another.

2. Backjumping in UDS

In usual solution searching, when the user’s new requirements arise, the search is restarted from the beginning using a larger constraint set (initial constraints plus new requirements). Or the search process backtracks and tries the nearest unfinished branch in a search tree. The former suffers from the problem that the results obtained are not used for the further search. The latter, which is called chronological backtracking, suffers from the problem that the choice to be revised may not contribute to the failure.
We will introduce a backjumping approach and apply it to our interactive problem solving of a truck dispatching task. Also we will study the case where dynamic rules can be created upon the user's request based on changes of practical situations. We will build a special menu where the user's queries may be input and automatically converted into deduction rules. These new deduction rules will be referred to in further search. If the user’s queries are his preferred solution pattern, as in the problem of closeness change and interaction, then the search can be effectively directed along the preferred directions.

3. Constraint recording

We also study an approach which can find and record the reason for a failed search. This can then be referred to in order to avoid the same failure in later search.

4. CSP techniques for pre-processing

We employ the CSP techniques to perform consistency checking in advance of performing the solution searching. This process is called pre-processing. Pre-processing is used for eliminating a number of potential values which are inconsistent with corresponding unary and binary constraints. Thus, we will make effective use of certain restrictions available from a set of constraints to construct a more limited set of possibilities. We expect that an amount of useless backtracking search can be avoided by this pre-processing. Some examples (in chapters 6 and 7) will be introduced to show how the search space can be considerably reduced and how system performance can be greatly improved using these CSP techniques. We also make use of CSP to improve efficiency in realizing UDS.

5. CSP techniques implementation in KEE

KEE [Intelllicorp 87] is a very powerful set of tools allowing problem solvers to solve artificial intelligence problems. It provides a number of programming
tools and techniques for building applications to represent and analyze knowledge. These tools are object-oriented programming, truth maintenance, KEE-worlds, rule system, active values and graphics. However, KEE does not offer the environment for implementing CSP techniques. Thus, it is inconvenient for the KEE to handle some problems where CSP techniques are necessary (as the CSP technique pre-processing in our cases here).

In this regard, we will implement a prototype CSP module called NAPCC (stands for Node, Arc, Path Consistency Checking) with the intention of creating a new environment to apply CSP techniques in KEE for general purpose. Therefore, such a module would be a convenient tool for the users to apply CSP techniques in various problem solving cases. Node, arc, path, unary and binary constraints in CSP will be represented by units and slots in KEE. All of the procedures and algorithms of CSP techniques built in KEE will be referred to as CSP module in the following text.

1.5 Overview of Thesis

The rest of the thesis is organized as follows.

In Chapter 2, we give a briefly description of the main features of KEE. Further we discuss KEEworlds and ATMS techniques, and use KEEworlds to express problem solving states.

In Chapter 3, we describe Constraint Satisfaction Problems (CSP). A CSP consists of a set of variables and a set of constraints. Each variable in a CSP is associated with a finite set of discrete values (domain). Each constraint limits the values the variables can take on. Basic techniques of CSP problem solving are node, arc and path consistency checking.

In Chapter 4, we establish a CSP module in KEE for general purpose programming using CSP techniques. In later chapters we will use this module in various applications to perform pre-processing and show how the CSP module can be used to effectively reduce the search space and to avoid some thrash behaviour (see section 3.2). The CSP module is designed in such a way that it constrains, a priori, the
number of potential values which are inconsistent with corresponding unary or binary constraints. In practical cases, the a priori constraining should be chosen with respect to both the specific problem to be solved and the specific method selected. The a priori constraining can improve the search efficiency in two ways:

- Pre-processing before searching. This is achieved by doing consistency checking. It can minimize the candidate values that are inconsistent with C. Therefore, the search space of the given problem may be reduced.

- Constraint recording during the search. We use constraints to check intermediate results. If they conflict with certain constraints, the dead-ends during search appear. The system stores the reasons for the dead-ends, so that the same conflicts will not arise again in the continuation of the search. The tree search efficiency is increased.

In Chapter 5, we present two strategies: (1) backjumping, (2) closeness change, to our interactive problem solving.

In the interactive problem solving, new requirements from the user can be given in the form of constraints. Such constraints are used to realise backjumping. That is, the search returns to the choice which causes failure and goes further based on the user's requirements. Thus, unnecessary backtracking can be avoided.

Our purpose is to establish a mechanism by which the solution search process can be effectively guided according to the user's preference. Special precautions must be taken in regard to making efficient use of the intermediate results obtained in further solution searching. We introduce here a scheme which we call closeness change which ensures that in the direction specified by the user, a closeness solution to the previous one will be found if it actually exists.

In Chapter 6, we establish a small system to offer UDS in the truck dispatching problem by using the techniques developed in Chapters 4 and 5.

In Chapter 7, we will construct a small prototype system in the domain of the crossword puzzle problem to demonstrate the basic strategies described in Chapters 4 and 5. These include using the CSP module established in Chapter 4 to perform data pre-processing, and employing user-directed search to achieve minimum change
from one solution to another solution. We will create a simple interface for user-directed search to make the procedure of solution search more friendly, efficient and controllable. With the intention of achieving efficiency, we make effective use of the information and results obtained in order to search further solution.

Chapter 8 provides a conclusion and an overview of the primary contributions.
Chapter 1: Introduction

In this chapter, we introduce a method by which the solution search process can be automatically guided according to the user's preferences. Special precautions are taken to ensure the user can view the intermediate results obtained during solution searching. We introduce here a scheme which we call closest solution searching, which requires that the solution specified by the user, a closest solution to the initial one can be found if it actually exists.

In Chapter 5, we describe a small system to offer UCS in the truck dispatching problem by using the techniques developed in Chapters 4 and 5.

In Chapter 7, we will construct a small prototype system in the domain of the vehicle scheduling problem to demonstrate the basic strategies described in Chapters 4 and 5. These include using the CSP module established in Chapter 4 to perform searching, and employing user-directed search to achieve minimum change...
Chapter 2

KEE, ATMS, KEEworlds

The Knowledge Engineering Environment (KEE) [Intellicorp 87] is one of the main knowledge system development tools which provide problem solvers with a set of programming tools and techniques for building knowledge engineering systems. It is an advanced hybrid developing environment which incorporates a number of well-proven artificial intelligence methodologies and techniques. As far as knowledge representation is concerned, KEE allows an expert system to be constructed with frame-based representation. It also provides rule-based programming, which allows the use of various inference mechanisms. Moreover, it provides object-oriented assertion, object-oriented programming, access to LISP, etc. In fact, KEE needs to be, and is being, improved to meet various requirements from practical expert system developments. One major effort made in this thesis study is, for example, to implant constraint satisfaction problem (CSP) techniques to the KEE environment. We have used these implanted CSP techniques to improve problem-solving efficiency in our case studies.

In this chapter, we introduce certain aspects of KEE which are relevant to the following chapters. Then we describe ATMS and its applications. Also we give a brief description about KEEworlds. In the following chapters, we will show how to solve practical problems based on KEE. We have incorporated a set of programming techniques to develop a paradigm in our interactive problem solver for user-directed search, and to effectively utilize information generated in the process of problem-solving.
2.1 KEE

KEE is actually a large set of well-integrated AI paradigms based on Common LISP language. It provides a wide variety of programming tools and techniques to represent and analyze knowledge and to solve knowledge system problems. It includes the production rules formalism which allows the use of various inference mechanisms, a frame-based language with inheritance, object-oriented programming paradigms with message passing, active values, KEEworlds and a truth maintenance system. The most significant advantage of KEE is that it integrates different programming methods and tools together. Another advantage is its way of organizing and aggregating knowledge into specific components (unit hierarchy) and of explicitly structuring the reasoning process. The third strong point lies in the power and the user-friendliness of its interface.

Figure 2.1 illustrates some of the packages and facilities of the KEE system built on Common Lisp. A higher package is based on the lower level packages and facilities.
2.1. Knowledge Representation

Knowledge representation in KEE is essentially based on a hierarchical network of frames with slot inheritance. Frames in KEE are called as units. The knowledge representation has the following characteristics:

1. Units can be used to represent objects of a specific problem domain. The units are grouped into a hierarchy from more general objects, called classes, to particular objects, called members. This facilitates inheritance of relevant attributes stored in slots from the higher level to the lower level.

2. A unit is made up of slots created to represent an object’s attributes. Slots can be used to represent two kinds of information: descriptive and procedural information. There are two kinds of slots: member slots and own slots.
   - Member slots are used for information that should be shared by all units in an object hierarchy.
   - Own slots are used to describe and represent attributes and properties of that slot alone. In other words, an own slot refers to a particular unit’s “own” properties.

3. Slots are composed of facets. Typical facets are inheritance role (controlling inheritance of slot values), valueclass (restrictions on the type of values a slot can have), and min.cardinality and max.cardinality (restrictions on the number of values a slot can have).

4. Units can have a procedural role and can enable behavioural models of objects and expertise to be built. Such objects are represented by a frame with methods (procedures) attached to the unit’s slots. Objects communicate with each other by sending messages. When an object receives a message, the message is interpreted, and the appropriate method is activated. Objects and message passing provide a natural way to represent entities in problems involving complex relationships.
5. Reasoning in KEE is based on rules, which are represented as units. The KEE rule system offers both forward and backward chaining as well as methods by which the two methods of reasoning may be mixed or linked in various ways.

In summary, knowledge representation is based on two components: facts and rules.

1. **facts** are frames with attributes and methods (in the sense used by object-oriented languages). Frames are organized into a taxonomy. Frames inherit properties (attributes) from the frames in the level above. Object-oriented programming is based on “methods” attached to slots; these are run when the slot receives a message of the form:

   \[
   \text{(unitmsg 'unit-name 'slot-name other.args.if.any)}
   \]

2. One kind of **rules**, namely, ‘same world action rules’, has the traditional production rule syntax:

   \[
   \text{If } \ (\text{premise}_1)(\text{premise}_2)\ldots(\text{premise}_{n1}) \text{ then } (\text{action}_1) (\text{action}_2)\ldots(\text{action}_{n2})
   \]

   Variables and Lisp function calls are allowed in both premise and action parts. The variables in a rule begin with a question mark. The variables are bound to values during the testing of the rule’s premises.

   An action of a rule consists of:

   - deleting a fact;
   - adding a fact;
   - modifying a fact;
   - executing a procedure (Lisp code);
   - invoking forward chaining on a different rule class from within forward chaining.

   KEE has enhanced the traditional production rule with the addition of new types of rules, such as new world action rules (for details see the next sub-section). Rules
are represented as frames, which make it possible to organize them into a hierarchy (taxonomy).

The KEE system contains several facilities for building and reasoning within a knowledge-based system. One facility for reasoning is based on rules. The KEE's rule system has two components: the rules and the inference mechanisms. Firstly, we describe the classifications of rules and their applicability. Then we discuss the mechanism available in the KEE rule system to perform data-direct reasoning (forward chaining). Here only forward chaining is presented since we employ it in the later chapters.

**Rule Type**

There are two types of rules in the KEE rule system: deduction rules and action rules. Deduction rules are used to establish dependencies between a conclusion and its premises. Action rules, on the other hand, are used to take actions. For deduction rules, the truth of the conclusion depends on the truth of the premises. If the premises become false, the conclusions also become false. For action rules, when the premises are satisfied, the actions specified in the conclusions are performed.

There are two kinds of action rules: same-world action rules and new-world action rules. The same-world action rules correspond to traditional production rules. They run in a world and perform actions in the same world. The new-world action rules create a new world in which the actions of the rule are performed. In other words, a new world action rule makes all the actions in the new world, while the parent world(s), where the premises were satisfied, remain(s) unchanged. The syntax difference between same world action rules and new world action rules is that new world action rules include the key world *in.new.world* (or *in.new.and.world*) which indicates new-world creation. *in.new.world* is used for rules which represent mutually exclusive alternative actions or choices. All the worlds created are under an exclusion set which is indicated by a black square with a cross in it. An exclusion set represents a set of worlds which are exclusive alternatives. The rules specified with the keyword *in.new.and.world* creates worlds that are not under an exclusion
Chapter 2: KEE, ATMS, KEEworlds

Figure 2.2: *in.new.world* creates worlds under an exclusion set.

Example:

\[
\begin{align*}
\text{(if } & \text{(the node.list of assist is } ?\text{nodes)} \\
& \text{(equal } ?\text{first (car } ?\text{nodes)}) \\
& \text{(the domain of } ?\text{first is } ?\text{v)}
\end{align*}
\]

\[
\text{then } \text{*in.new.world*}
\]

\[
\text{(the value of } ?\text{first is } ?\text{v)}
\]

The above rule is a simple rule for finding an assignment to a variable in the crossword puzzle problem. *?nodes, ?first, and ?v* are variables. When this rule can be applied in a given world (that is, the premises of this rule are proven *true*), say in world *start*, an exclusion set is created. Each subsequent application of the rule in *start* causes a new world, which falls under the same exclusion set to be created (see Figure 2.2). The world *node1-sound* and *node1-sails* were created by applying the above rule with two different bindings for *?v*, namely, ‘sound’ and ‘sails’, and represent mutually exclusive choices that *?first* (bound by node1) might take. This results in that each node can only be assigned to one value at any one time. The worlds that represent different assignment values of the node are mutually exclusive. If the above two worlds are merged, the resultant world becomes inconsistent. So merging any two worlds is avoided. Each inconsistent world is indicated by a black square in it (Figure 2.2). We use this kind of rule to control the reasoning and searching process.

Rules can be expressed either in the English-like form or the prefix-form. They are represented as units and classified in a hierarchical way like any other unit. Thus, we can control which rules should be fired. This feature makes it possible to
organize and select sets of rules for solving specific problems.

**Forward Chaining**

KEE supports a mixed-strategy approach by allowing both forward and backward chaining to be applied to the same problem. It allows the user to select a conflict-resolution strategy from a number of options, including least premise complexity, greatest premise complexity, least weight, and greatest weight.

Here we explain how forward chaining works because we use forward chaining to solve our problems. The forward chaining cycles as follows:

1. Evaluating rule premises one by one;
2. Adding rules successfully instantiated to agenda;
3. Selecting an instantiated rule to apply;
4. Taking the actions required by the rule.

This cycle is continued until the agenda is empty.

### 2.2 Assumption-Based Truth Maintenance System (ATMS)

Truth maintenance is a means of keeping track of beliefs and their dependencies developed during an inferencing process [McAllester 90]. The KEE system includes a system called the Assumption-based Truth Maintenance System (ATMS) which provides facilities for setting up and maintaining dependencies between facts. Dependencies between facts set up in the ATMS are called *justifications*.

A justification consists of two main components:

- **justifiers** – these are the facts that give evidence for the truth of justificand. Justifiers are antecedents.
- **justificand** – this is a fact that is only **true** while all the justifiers are **true**. Justificands are consequents.
2.2.1 Deduction Rules and Justifications

Justifications can be established by creating and applying deduction rules. A deduction rule has the following form:

\[
\text{while } (\text{premise}_1)(\text{premise}_2)\ldots(\text{premise}_{n_1}) \\
\text{believe } (\text{conclusion}_1) (\text{conclusion}_2) \ldots (\text{conclusion}_{n_2})
\]

When a deduction rule is applied in forward chaining mode, one justification for each conclusion is set up for each set of true instantiated of the premises. All the premises are justifiers and the conclusion is the justificand.

2.2.2 Constraints

The special fact false can be the conclusion of a deduction rule. Thus, we use deduction rules to establish the fact false in any world which violates certain constraints. Therefore, we use deduction rules to represent constraint rules and detect contradictions. For example, in the Truck Dispatching Problem there are a number of constraint rules, such as 'you can't put more on a truck than it can hold'. The corresponding deduction rule for expressing this constraint is as follows:

\[
\text{while } (\text{the max. weight of } ?\text{trip is } ?m) \\
\text{(the truck of } ?\text{trip is } ?t) \\
\text{(the weight.capacity of } ?t \text{ is } ?w) \\
\text{(lisp } (> \ ?m \ ?w)) \\
\text{believe false)}
\]

One of dependencies was created by applying the above deduction rule with bindings trip.1, 600, T4, 500 for ?trip, ?m, ?t, ?w respectively. This dependency will be represented by a justification which states that while the facts

\[
\text{(the max.weight of trip.1 is 600)} \\
\text{(the truck of trip.1 is T4)} \\
\text{(the weight.capacity of T4 is 500)}
\]
the fact false is deduced. That is, when the premises are true in any world, its conclusion false is added to that world which thus becomes inconsistent. If the facts in the premises are removed, the conclusions are automatically deleted from the knowledge. The reasoning process ignores inconsistent worlds in further reasoning. We can use the justifications of contradictory conclusions to determine which assumptions are mutually contradictory. One of the purposes of establishing and maintaining these dependencies is that if a contradiction is detected, we allow backtracking along the reasoning, changing the assumptions in some way, and reasoning forward from there.

2.3 KEEworlds

KEEworlds is a facility provided by the KEE system to allow representation of alternative states of situations. It is used for modelling and exploring different hypothetical situations which are represented by worlds. For example, in one world, node1 is assigned by ‘sound’; in another world, by ‘sails’; and so on. We employ KEEworlds and new world action rules for constructing and representing solutions through a step-by-step process, with each new world representing a decision in the search.

Some of the uses of the KEEworlds facility are to:

- try out different values for ‘own’ slots in different worlds;
- run rules in worlds, without changing any slot values in the original state of the knowledge base; and
- run reasoning processes, using worlds to record the intermediate stages of the processes.

KEEworlds is one of the useful facilities for managing problem solving which involves search since it can generate and store the intermediate results in the search space. If the search meets a dead end, the search backs up to the intermediate result and continues the search from that point. We use appropriate rules and ATMS|B to design different backtracking strategies.
We shall use worlds to represent the search process. So we give a brief description of the worlds here.

### 2.3.1 Worlds

The based structure for modelling the searching process is a directed acyclic graph of *worlds*. Each world may be regarded as representing an individual specific action, or state change. Each successor of a world in the graph models alternative actions.

Figure 2.3 shows an example world graph from a truck dispatching problem. ‘+trip.1, trip.2’ beside *world 0* means that two trips need to be allocated feasible trucks and drivers. ‘+t1, t2’ and ‘+d1, d2’ means that these candidate trucks and drivers are available at an initial state. The additions at *world 0* produce the initial state, while the addition and deletion at *world 1* represent the action of assigning *t1* to *trip.1*. The addition and deletion at *world 2* represents the effect of assigning *d1* to *trip.1*, while those at *world 3* correspond to the alternative effect of assigning *d2* to *trip.1*. ‘-trip.1’ means that *trip.1* is removed from the trip list because a truck and a driver were already assigned to it. ‘-t1, -d1’ means that *t1, d1* are removed from the candidate trucks and drivers respectively. The condition of the actions is tested in the parent *world 1* before *world 2* and *world 3* were to be constructed.

Only the *effects* of actions are recorded in the world. Thus, each world corresponds to a decision point in the search. A branch of the world graph corresponds to a linear sequence of actions during the course of the search.

A world can contain the following facts:

1. facts inherited from ancestor worlds;
2. direct additions at this world;
3. deductions (performing action part of the rule) from the facts in 1 and 2.

The deduced facts may include the specific fact *false*, representing a contradiction. A world which contains *false* or where it can be deduced is an inconsistent world which represents an inconsistent state of knowledge. We avoid such a world in further reasoning.
Figure 2.3: A world graph from truck dispatching problem.
2.3.2 Worlds and Forwarding Chaining

Forward chaining can run new world action rules in a particular world and the actions of rules can be performed in a new world which is a child of the world. A world can be used for expressing problem solving states, and these states can be used for preserving search state during the course of solution searching.

We use new world action rules to search for a solution in a forward chaining mode. This is a process of finding a sequence of actions from an initial state to a desired state. Each world created during the course of reasoning represents a corresponding decision in the search tree.

2.3.3 Worlds and Justifications

In general, problem solving which involves searching leads to the discovery of some set of beliefs. In such systems, different sets of beliefs are believed at different stages during the course of problem solving. Each world in KEEworlds is used to express different points (different sets of beliefs) in the search.

Justifications can be created by deduction rules. Once a justification has been established, it applies in all the worlds. If a set of facts in a world conflicts with a set of constraints, the fact false can be introduced into the world. An inconsistent world is a world that contains the fact false. An inconsistent world is shown with a black square in it. We can use deduction rules to introduce the fact false into any world that violates certain constraints. Once a world has become nogood, the system will ignore it for further consideration. Thus, an inconsistent world expresses the dead end search in the search tree. When a world becomes inconsistent, any existing child worlds of that world also become inconsistent.

Thus, because of its many advantages, we use KEE from which we can gain considerable power and flexibility when constructing a representation of our domain knowledge and controlling the search process to solve our practical problems. Furthermore, KEE is an open architecture which is an important factor in extending KEE. This is useful for ensuring KEE's flexibility and ability to evolve. Therefore, we can extend KEE, which is oriented towards handling of specific AI problems. In
Chapter 5, we develop NAPCC package (see Chapter 4) in KEE to realize CSP techniques. Two case studies in Chapters 6 and 7 will take advantage of it to improve search efficiency.

Chapter 3

Constraint Satisfaction Problem

Techniques

3.1 Introduction

Many artificial intelligence tasks can be described as constraint satisfaction problems (CSP) [Chapter 47]. To solve a CSP is to find values for a set of variables which are subject to a set of constraints given in the problem under consideration. A constraint may affect one, two or more domains in the problem and thereby restrict the corresponding variable to take values in a certain range which is generally smaller than the original range.

In recent years, constraint satisfaction problem solving has received considerable research interest in artificial intelligence [Dechter 86] and [Minton 86]. The well-known constraint propagation techniques, originally were advocated by [Wallace 76], [Mackworth 77] and [Montanari 80]. The usual procedure for solving a CSP is backtracking. The most obvious advantage of backtracking is its simplicity and its universal applicability. The disadvantage is its potential inefficiency. That is, a number of useless backtracks ("dismaying behaviour" [Mackworth 77]) may occur in a particular CSP. This means that the search space is larger than necessary for solving the problem. Thus, some special techniques are introduced for some CSP problems to yield a smaller search space and to avoid dismaying behaviour. Node, arc, and path
Chapter 2: KEE, ATMS, KEEworlds

2.5.2 Worlds and Justifications

In general, problem solving which involves searching leads to the discovery of some set of beliefs. In such systems, different sets of beliefs are believed to be held at different stages during the course of problem solving. Each world in KEEworlds is used to express different points (different sets of beliefs) in the search.

Justifications can be created by deducing rules. Once a justification has been established, it applies to all the worlds. If a set of facts in a world conflicts with a set of constraints, the fact false can be introduced into the world. An inconsistent world is a world that contains the fact false. An inconsistent world is shown with a black square on it. We can use deduction rules to introduce the fact false into any world that violates certain constraints. Once a world has become negated, the system will ignore it for further consideration. Now, an inconsistent world expresses the dead end search in the search tree. When a world becomes inconsistent, any selecting child worlds of false would also become inconsistent.

This, because of its many advantages, we use KEE from which we can gain experience power and flexibility when constructing a representation of our domain. This, also, is an important factor in extending KEE. This is why we say KEE's flexibility and ability to evolve. Therefore, we use actual KEE's object-oriented approach handling of specific AI problems.
Chapter 3

Constraint Satisfaction Problem Techniques

3.1 Introduction

Many artificial intelligence tasks can be described as constraint satisfaction problems (CSP) [Shapiro 87]. To solve a CSP is to find values for a set of variables which are subject to a set of constraints given in the problem under consideration. A constraint may affect one, two or more domains in the problem and thereby restrict the corresponding variables to take values in a certain range which is generally smaller than the original range.

In recent years, constraint satisfaction problem solving has received considerable research interest in artificial intelligence [Dechter 88] and [Minton 90]. The well-known constraint propagation techniques, originally were advocated by [Waltz 75], [Mackworth 77] and [Montanari 83]. The usual procedure for solving a CSP is backtracking. The most obvious advantage of backtracking is its simplicity and its universal applicability. The disadvantage is its potential inefficiency. That is, a number of useless backtracks ('thrashing' behaviour [Mackworth 77]) may exist in a particular CSP. This means that the search space is larger than necessary for solving the problem. Thus, some special techniques are introduced for some CSP problems to yield a smaller search space and to avoid thrashing behaviour. Node, arc, and path
consistency checking are considered as useful techniques in this regard. Using these techniques can have a great effect on reducing the size of the search space. Some examples will be introduced later to show how the search space can be considerably reduced by using these CSP techniques.

CSP techniques are useful in many applications. The well-known examples of application of CSP techniques are the traditional puzzle-solving problems [Mackworth 87], map-coloring problem [Freuder 78], understanding line drawings [Montanari 74], electronic circuit analysis [Kelly 82] etc. The applications of CSP techniques range from database retrieval to scene analysis. CSP techniques may be used to prevent unnecessary search thereby leading to solution strategies that do not require exponential time for particular task domains [Mackworth 77]. We study the application of CSP techniques such as node, arc, and path consistency checking to process primary data prior to solution search. We view this process as pre-processing. Pre-processing is used for eliminating a number of potential values which are inconsistent with corresponding unary or binary constraints. Thus, we will take advantage of some restrictions available from the unary and binary predicates to construct a more limited set of possibilities. By this pre-processing, we can expect that an amount of useless backtracking are avoided.

In this chapter, we will first introduce basic definitions of CSP. Issues in solving a CSP by backtracking are discussed. The constraint network for representing CSP is described. Also, the basic techniques used in CSP to avoid thrashing behaviour are presented. Finally, with an example, we demonstrate how the search space of a problem is reduced.

### 3.2 Constraint Satisfaction Problems Definition

Following existing research works on CSP [Mackworth 87], we perceive the CSP as a set of variables and each variable has a domain of values,

\[
V = \{v_1, v_2, \ldots, v_n\} \text{ a set of variables;} \\
D = \{D_1, D_2, \ldots, D_n\} \text{ a set of domains;} \\
D_i = \{d_{i1}, d_{i2}, \ldots, d_{im_i}\} \text{ a set of domain values.}
\]
Every variable $v_i$ must be assigned a value $d_{ij}$, where $d_{ij} \in D_i$, and such an assignment will be denoted by $v_i \leftarrow d_{ij}$.

A CSP also consists of a set of constraints. A constraint $C_i(v_i_1, \ldots, v_i_n) (v_i_1, \ldots, v_i_n \in V)$ is a subset of Cartesian product $D_{i_1} \times \cdots \times D_{i_l}$ ($D_{i_1}, \ldots, D_{i_l} \in D$). These constraints determine whether assignments of values to variables are compatible with each other or not. In other words, these constraints taken together constitute a global constraint which specifies which sets of values $a_1, a_2, \ldots, a_n$ for $v_1, v_2, \ldots, v_n$ can simultaneously satisfy all the given constraints. A solution to a CSP is an n-tuple assignment of values to all variables which satisfy all the constraints for the problem to be solved.

Various restrictions on the general definition of a CSP are possible. One restriction is that the domains may be required to have a finite number of discrete values. Another may further require that all the relations be unary or binary constraints. That is, they only constrain individual variables or pairs of variables. The unary constraint on individual variable $v_i$ and the binary constraint on two variables $v_i$ and $v_j$ are expressed as $P_i$ and $P_{ij}$ respectively. A binary constraint $P_{ij}$ between two variables $v_i$ and $v_j$ is a subset of the Cartesian product of their domains:

$P_{ij} \subseteq D_i \times D_j$.

We will restrict our attention to unary and binary constraints. This is partly for simplicity, and partly because a non-binary constraint can be converted into a binary constraint by the introduction of additional variables.

In general, one may represent a CSP as being equivalent to determining the truth value of a formula in first-order predicate logic:

$$(\exists v_1)(\exists v_2) \cdots (\exists v_n)(v_1 \in D_1)(v_2 \in D_2) \cdots (v_n \in D_n)$$

$$P_1(v_1) \land P_2(v_2) \cdots \land P_n(v_n)$$

$$P_{12}(v_1, v_2) \land P_{13}(v_1, v_3) \land \cdots \land P_{n-1,n}(v_{n-1}, v_n) \quad (1)$$

the binary constraint $P_{ij}$ is true if there do not exist constraints between two variables $v_i$ and $v_j$. This is the same for a unary constraint $P_i$: $P_i$ is true if no constraint is imposed on $x_i$. 


Chapter 3: Constraint Satisfaction Problem Techniques

A large number of seemingly different problems can be formalized as CSP problems [Shapiro 87]. Of particular theoretical interest is the graph-colouring problem, which for example, considers whether three colours suffice to colour a given planar graph such that each vertex is a different colour from each of its neighbours. This can be formulated as a CSP by creating a variable for each vertex to be coloured, associating with each variable the domain \{ red, green, blue, white, \cdots \}. The binary constraint is that if two vertices are connected by an edge they do not have the same colour. A unary constraint on each vertices requires it to be one of three colours, namely \{ red, green, blue \}. The famous “four colour problem” can also be represented in these terms.

A CSP is a general problem involving searching. A simple and direct approach to solve a CSP is ‘generate-and-test’. All the variables domains have finite number of discrete values so that the assignment space \( D = D_1 \times D_2 \times \cdots \times D_n \) is finite. And so one may evaluate the formula (1) on each element of D and stop if it is evaluated to be true. But this approach of testing every possible combination of values faces an obvious combinatorial explosion. One standard approach that can be used to solve a constraint satisfaction problem is a depth-first search. The search process fixes values to variables as long as it can find a value for each variable that is compatible, according to the constraints \( C \), with the values already fixed to previous variables. Whenever the procedure cannot find a value for a new variable, it backtracks to the previous unit and tries to find an untried value for that variable. The efficiency gained from backtracking search arises from the fact that a potentially very large subspace of D, namely, the product space of the currently unassigned variable domains, is eliminated from further consideration by the single predicate failure. This simple technique can find the solution if it exists, but the time is taken to find a solution tends to be exponential in the number of variables both in the worst-case and on average [Mackworth 84].

On the other hand, backtracking search procedure may be quite inefficient because it suffers from thrashing behaviour [Mackworth 87]. Thrashing behaviour can be defined here as the repeated exploration of subtrees of the search tree that differ
only in inessential features, such as the assignments to variables irrelevant to the failure of the subtrees. That means that a poor choice of values for one of the first variables can cause failure of all paths stemming from that choice. Let us consider a simple example in detail to clarify this. Assume \(v_1\) and \(v_2\) range over the domain \{1, 2, 3\}, variable \(v_k\) ranges over \{1\}, and there are a set of components \(v_3, \ldots, v_{k-1}\). Some of the constraints are the conditions \(C_1\), stating that the values of \(v_1\) and \(v_2\) have to be different, and \(C_k\), that \(v_k\) has to be different from \(v_1\). Suppose the variables are instantiated in the order \(v_1, v_2, \ldots, v_k\). A backtrack search (depth-first) starts with \(v_1 = 1\), and because of \(C_1\), \(v_2 = 2\), then an assignment for \(v_3, \ldots, v_{k-1}\) is searched for. But attempting to find a value for \(v_k\) fails due to \(C_2\). At this point an exhaustive search over the sub-search tree \(v_3, \ldots, v_{k-1}\) is started. That is, the search will try all the combinations of values for \(v_3, \ldots, v_{k-1}\) before finally discovering that the value 1 is not possible value for \(v_1\). The backtracking search among \(v_3\) to \(v_{k-1}\) is useless. Figure 3.1 clearly illustrates the search space of this problem, where \(\times\) is dead-end in search tree. In order to avoid this poor behaviour, we should eliminate the value 1 from variable \(v_1\) domain. Thus the search inside the dotted box in Figure 3.1 never appears during the course of searching process.

Now we have shown the problems existing in backtracking search. The poor behaviour can have a dramatic effect on the size of the search space and time. To make the depth-first search more efficient, we must eliminate those paths which terminate because they are not contained in any feasible solutions. This research has led up to the current great interest [Mackworth 87]. Many of the techniques are designed to reduce, eliminate or prevent thrashing behaviours.

Thus, to do an efficient search, we must first remove from \(D = \{D_1, D_2, \ldots, D_n\}\) many values which do not participate in a possible solution for solving a CSP. That is, we rule out some values in variables which are inconsistent with corresponding unary and binary constraints. Therefore, this process could simplify subsequent backtrack search and reduce the search space.

In the following section we will introduce a network representation of the constraint satisfaction problem and some basic methods to eliminate thrashing be-
haviour. The process of reducing thrashing behaviour will be done by performing consistency checking.

### 3.3 A Network Representation of a CSP

It is convenient to represent the CSP task specification as a constraint network. A constraint network is a directed graph. It consists of the following components:

1. **Node**: Each node represents the variable. Each node associates a set representing the variable’s domain and the corresponding unary constraint on it.

2. **Arc**: Binary constraints are represented by a labelled, directed arc. An arc from node i to j corresponds to $P_{i,j}$. There is an arc connecting two nodes if there does exist binary constraint to these two nodes.
3.4 Inconsistency and Consistency

For instance, the above-mentioned graph-colouring problem can be formulated as constraint network (shown in Figure 3.3), where three vertices are considered and the domains of all the variables contain the same values, namely \{ red, green, blue, white, violet \}.

The nodes \( v_1, v_2, \) and \( v_3 \) include the possible values \{ red, green, \ldots \} for each variable respectively and the relevant unary constraints. There exist unary constraints on all the nodes in the network due to limiting all the variables to three colours, namely \{ red, green, white \}. The arcs between \( v_1 \) and \( v_2 \), between \( v_2 \) and \( v_3 \) and so forth correspond to the binary constraints of the graph-colouring problem. Any CSP problem can be represented as a constraint network.

3.4 Inconsistency and Consistency

We give here a detail description of such sources which can cause thrashing behaviour. Then, the definitions of consistency and inconsistency are presented as a means for addressing the problem of thrashing behaviour. Finally, we introduce some methods which can be taken to prevent this poor behaviour.

1. Consider a unary constraint on one variable: If a value \( x_j \) from the domain \( D_i \) for variable \( x_i \) does not satisfy \( P_i(x_j) \), which is called a node inconsistency, it will be the cause of useless assignment to other variables. The value \( x_j \) is not possible value for \( v_j \). Therefore, the value \( x_j \) could be ruled out from domain \( D_i \) once and for all. We could delete all the values from domains that
do not satisfy the corresponding unary constraints. The search space could be cut down via reducing variable domains. This process can be viewed as node consistency checking.

2. Consider binary constraint on one arc: Assume that for a value $l_i$ in $v_i$, there does not exist any value $d_{jk}$ in $v_j$ such that the corresponding binary constraint $P_{i,j}(l_i, d_{jk})$ holds. And suppose the variables are assigned in the order $v_i, v_{i+1}, \ldots, v_j$ ($j > i$). Backtracking will try all the values in $v_j$ and fail. It then will try all the values in $v_{j-1}$ and so on. Finally the search discovers that $l_i$ is not a suitable value for $v_i$. The search among $v_{i+1}, \ldots, v_{j-1}$ is fruitless. This can be illustrated in Figure 3.1.

3. A third source which causes inefficiency and replication of effort occurs in following situation. Suppose $v_i = a, v_j = b$, a and b together satisfy the corresponding binary constraint $P_{i,j}$. And suppose for the variable $v_k$ there exists binary constraints $P_{i,k}, P_{k,j}$. But there does not exist any value $x_k$ in $v_k$, such that $P_{i,k}(a, x_k), P_k(x_k)$ and $P_{k,j}(x_k, b)$ are simultaneously satisfied. Thus value pair $(a, b)$, a in $v_i$ and b in $v_j$ must not appear in any possible solution for a given problem. As in the previous case, this may be rediscovered many times by a backtracking search process.

We introduce the definitions about consistency and inconsistency in a constraint network. Consistency means that all the domain values are consistent with the corresponding unary and binary constraints for the specific constraint network. Consistency for the network includes three kinds, namely node, arc, path consistency. Here we give the definition of node consistency, arc consistency and path consistency.

(i). Node consistency

Node $i$ is node consistent if and only if for any value $x \in D_i$, $x$ satisfies the unary constraint $P_i(x)$.

(ii). Arc consistency

Arc $(i, j)$ is arc consistent if and only if for any value $x \in D_i$, there is a value $y \in D_j$, such that $x$ and $y$ satisfy the binary constraint $P_{i,j}(x, y)$. 
(\forall x)(x \in D_i) \rightarrow (\exists y)(y \in D_j) \land P_{ij}(x, y)

(iii). Path consistency

A path of length m through the node \((i_0, i_1, \ldots, i_m)\) is path consistent if and only if for any values \(x \in D_{i_0}\) and \(y \in D_{i_m}\) such that \(P_{i_0}(x) \land P_{i_m}(y) \land P_{i_0i_m}(x, y)\), there exists a sequence of values \(z_1 \in D_{i_1}, \ldots, z_{m-1} \in D_{i_{m-1}}\) such that

(i). \(P_{i_1}(z_1) \land \ldots \land P_{i_{m-1}}(z_{m-1})\);

(ii). \(P_{i_0i_1}(x, z_1) \land P_{i_1i_2}(z_2, z_3) \land \ldots \land P_{i_{m-1}i_m}(z_{m-1}, y)\).

Node, arc and path is inconsistent if the condition of consistency is not satisfied.

In a word crossword puzzle problem, for instance, assume a variable domain \(D_1 = \{\text{look, talk}\}\) and unary constraint to \(v_1\) is that the length of word in \(v\) is 4. The \(v\) is node consistent. If \(D_1 = \{\text{look, say}\}\) and the same restriction to \(v_1\), \(v_1\) is node inconsistent because the value ‘say’ in \(D_1\) violates the constraint to \(v_1\).

The aim of consistency checking is to remove inconsistency from constraint network which can never be part of any possible global solution. Consistency checking can be seen as making the constraint network consistent. It includes three levels of consistency checking, namely node consistency checking, arc consistency checking and path consistency checking.

### 3.4.1 Node Consistency Checking

The simplest consistency checking is node consistency checking. In graph-colouring example, the potential domain of values for \(v_1, v_2, \) and \(v_3\) is given as \{red, green, blue and white\}, but the unary constraints on \(v_1, v_2, \) and \(v_3\) require them to red, and green. We can immediately eliminate blue and white from all the nodes in constraint network. In this way some domain values which are inconsistent with problem constraints can be eliminated. Thus, the search space is reduced.

A constraint network is called a ‘node consistent network’ if and only if every node of its network is consistent. Therefore, node consistency checking must do node consistency checking on all the nodes of network. Since node consistency is concerned only with unary constraints, there is no interaction between the nodes. Thus, node consistency checking is easy to realize. One pass over all the nodes in
constraint network for node consistency checking can make the constraint network node consistent. Detailed processing is described in a later section.

3.4.2 Arc Consistency Checking

The second consistency checking is arc consistency checking. The arc from variable \( v_1 \) to variable \( v_2 \) is inconsistent if for a value in \( v_1 \), namely \( a_1 \), there does not exist any value \( a_2 \) in \( v_2 \) such that \( a_1 \) and \( a_2 \) together satisfy the corresponding binary constraint between \( v_1 \) and \( v_2 \). For instance, suppose that the possible values for \( v_1 \) and \( v_2 \) are the same, that is \( D_1 = D_2 = \{ \text{laser, hoses, house, steer, sheet} \} \). The constraint between \( v_1 \) and \( v_2 \) is that the third letter of word in \( v_1 \) is the same as the first letter of word in \( v_2 \). The arc from \( v_1 \) to \( v_2 \) is inconsistent because for a value \{ \text{steer} \} in \( v_1 \), there is not any value \( w \) in \( v_2 \) such that \{ \text{steer} \} and \( w \) satisfy the relation “the third letter of steer is the same as the first letter of word \( w \) in \( v_2 \)”. To make this arc consistency, remove steer from \( v_1 \) and similarly \{ house, sheet \} from the \( v_1 \) domain. This certainly cuts down the search space.

The network is arc consistent if and only if all the arcs are consistent. We perform arc consistency checking to make all the arcs in constraint network arc consistent. Since arc consistency is concerned with the binary constraints, there is interaction among the nodes. Thus, a single pass of arc consistency checking over all the arcs in the network will not guarantee that the network is arc consistent. The strategy adopted to handle this efficiently will be presented in a later section. It should also be noticed that it is entirely possible for a network to be arc consistent, and still no solution exists.

Nevertheless, it may be helpful to remove arc inconsistencies for a constraint network. Generally, most of nodes will have a small set of values remaining after arc consistency checking. Thus, finding values for a set of variables which simultaneously satisfy all the constraints can be quickly found with a small tree search.
3.4.3 Path Consistency Checking

The third consistency checking is *path consistency checking*. A path of two length from \( v_1, v_2, \) to \( v_3 \) is path inconsistent if there is a pair of values \( a \) from \( v_1 \), \( c \) from \( v_3 \) such that \( a \) and \( c \) satisfy the corresponding unary constraints, and \( a \) and \( c \) together satisfy the binary constraint between \( v_1 \) and \( v_3 \), while there does not exist any value \( b \) in \( v_2 \) such that \( a, b, c \) can simultaneously satisfy constraints \( P_{12} \) and \( P_{23} \).

To remedy this path inconsistency, remove the \( (a, c) \) pair from all potential pairs between \( v_1 \) and \( v_3 \).

In order to improve efficiency, it is necessary to cut down the search space and avoid classes of thrashing behavior by eliminating combinations of values which could not appear together in any set satisfying all the constraints. These combinations are eliminated – this can be viewed as removing inconsistencies in a constraint network representation of the problem. This process can be viewed as consistency checking.

3.5 Summary

In this chapter, we have described the concepts of CSP and basic CSP techniques – node, arc and path consistency checking. Consistency checking can remove local inconsistencies which can never be part of any feasible solution from the constraint network and, therefore, can reduce the search space. However, it cannot guarantee that the set of values remaining satisfy all the constraints in the specific problem under consideration. It may even fail to reveal that no solutions of the given CSP problem at all exist. Thus further search is usually required to find the solution that consists of the acceptable combinations of values.

CSP techniques has been considered very useful for some particular tasks to improve search efficiency and effectiveness. In some specific applications, for example, consistency checking may greatly facilitate the search for solutions, because it can provide a much smaller search space for further search [Dechter and Meiri, 1989]. In our case studies which are carried out on KEE, we will show how to use CSP techniques to improve search efficiency. Based on the usefulness of using CSP techniques
demonstrated in our cases, we will describe how CSP techniques are introduced and how an independent CSP module is implemented in KEE.
Chapter 4

Implementation of CSP Techniques in KEE

4.1 Introduction

For many practical cases, CSP techniques can be chosen with regard to both the specific problem to be solved and the specific method which has been selected. This research has received considerable attention in recent years in the context of work on AI applications [Shapiro 87].

From chapter 2 we know that KEE is a knowledge system development product that provides a set of powerful and efficient programming tools and techniques for building applications to represent and analyse knowledge to solve problems. However, KEE does not provide the environment for handling CSP techniques. Thus, it will be inconvenient for KEE to handle some AI tasks where CSP techniques are necessary. In this regard, we have studied the basic CSP techniques and their relevant properties with the intention of creating a new environment in KEE to achieve CSP techniques.

The basic CSP techniques include node, arc, and path consistency checking. This consistency checking is achieved by removing values from domains which do not satisfy corresponding unary and binary predicates. This process can allow for a subsequent search in a smaller space. We will use the basic CSP techniques — node,
Chapter 4: Implementation of CSP Techniques in KEE

arc and path consistency checking — to perform data processing and to improve search efficiency. From these cases, we show that these CSP techniques are very useful for the improvement of search efficiency in many applications. In this matter, we have created an independent CSP module in KEE. Hence the CSP techniques are made available for being used in non-CSP expert systems, e.g. rule-based systems on KEE.

The basic CSP techniques implemented in KEE is called as NAPCC (which stands for Node, Arc, Path Consistency Checking) module. The purpose of the NAPCC module in KEE is to create a constraint network and to make the network consistent. That is, NAPCC can perform node, arc and path consistency checking.

To use the NAPCC, the user needs to define explicitly nodes, the corresponding domain and unary predicates, arcs and the relevant binary predicates and paths which are used in consistency checking.

In this chapter, we first discuss how to represent node, arc and relevant knowledge for CSP in KEE. Then we describe how to build and to implement a node-arc-path consistency checking (NAPCC) module to realize the basic CSP techniques by utilizing KEE’s object-oriented programming capability and other useful tools and techniques in KEE. Finally we show how to use the NAPCC and discuss its various applications. We will show in the case studies in the following chapters as practical examples of using NAPCC.

4.2 CSP Representation in KEE

We first give the definitions needed to state the CSP problem precisely.

Definition 4.1 A constraint satisfaction problem (CSP) is a structure < V, D, P1, P2 > where

1. V is a set of variables. V = {v1, ..., vn}, where v1, ..., vn may take on values from a set of domains;

2. D is a set of domains of ‘individual’ variables; D = {D1, ..., Dn}, where each domain is discrete and finite domain;
3. $P_1$ is a set of unary relations on $V$;

4. $P_2$ is a set of binary relations on $V$. $P_2 \subseteq \{D_i \times D_j|i \neq j\}$.

CSP problems can be represented as a constraint network, which consists of nodes, arcs, and paths.

**Definition 4.2** A node in a constraint network is a structure $<\text{Name}, \text{Domain}, p_i>$ where

1. Name is a name of node given by the user;

2. Domain consists of a finite number of discrete values which the variables may take;

3. $p_i$ is a unary constraint on Domain, $p_i \in P_1$.

Here we represent $p_i$ as a lambda expression or as a function name which must be written in Common Lisp, KEE functions, or combination of both. If a function name is used, the function definition should be given. If a value $x$ satisfies $p_i(x)$, then $p_i(x)$ returns true, otherwise it returns false. The node consistency checking (discussed in the next section) employs this information to decide which values can be eliminated from the corresponding variable domain.

**Definition 4.3** An arc in a constraint network is a structure $<\text{Name}, \text{Pair}, p_{ij}>$ where

1. Name is a name of an arc given by the user;

2. Pair consists of two nodes, where one is the origin node of the arc and the other is the destination of the nodes. These are all associated with a particular arc;

3. $p_{ij}$ is a binary constraint on arc, $p_{ij} \in P_2$.

where $p_{ij}$ is represented as a lambda expression or a function name in the KEE system. If a pair of values $x$ and $y$ satisfy the corresponding binary relation $p_{ij}(x, y)$, then $p_{ij}$ returns true, otherwise it returns false. The arc consistency checking utilizes this information to decide which values can be ruled out from the corresponding variable domain.
Definition 4.4 A path in a constraint network is a structure < Name, Nodelist > where

1. Name is a name of path given by the user;
2. Nodelist consists of all the nodes along the path.

We can represent and manage a model of a CSP, including the objects, relationships, and behaviours about the given CSP via KEE's flexible and expressive frame system.

In the following, we give a detailed description of how to represent a given constraint network in KEE. We present three class units: NODES, ARCS, and PATHS units. These units are used to store and analyse information which is required to perform consistency checking.

4.2.1 NODES unit

We use units (frames) to express the object of a given problem and arrange these units in class hierarchy via KEE frame system. The representation of this hierarchy is similar to the "class" and "member of class" way of organizing objects. We represent NODES as the "class" of all nodes in constraint network. We then describe all the nodes as "member of class" NODES (as shown in Figure 4.1). Both class and member classes are represented as units. This hierarchical organization, coupled with the inheritance mechanism, is an efficient way of storing, retrieving, analyzing knowledge and reasoning with it.

The description of the attributes of each of the objects is accomplished with slots within a unit. Slots are used to represent two kinds of information:

- *descriptive or factual* information, such as the values of node’s domain.
- *procedural* information, in the form of LISP, such as a program to decide whether a value in one node satisfies the corresponding unary constraint. Method slots are used for containing procedural information in KEE system.
4.2. CSP Representation in KEE

Figure 4.1: The representation of NODES unit and its descendants

In order to distinguish between local and general information, there are two kinds of slots: member slots and own slots. Member slots can be inherited, while own slots cannot be inherited.

1. Member slot:

A member slot describes an attribute shared by all members of a class. Member slots occur only in class units. For instance, it is necessary to associate "add values" and "delete values" with all the nodes in a constraint network. Therefore, the class unit NODES of a CSP knowledge base contains member slots named ADD.VALUES and DELETE.VALUES (see Table 4.1).

In Table 4.1, Inheritance determines how slot values are passed down through object hierarchies. The method slots which contain Lisp code (object-oriented behaviour) must use the method inheritance roles. Valueclass restricts on the type of values a slot can have. For method slot (attaching procedural behaviour to the slot), method attaches to 'valueclass'. If we limit the values of a slot to any number, the number attaches to the valueclass (Table 4.1).

All the members of NODES unit receive the ADD.VALUES and DELETE.VALUES by means of inheritance (as shown Table 4.2). In Table 4.2, we can see that NODE1 inherits ADD.VALUES and DELETE.VALUES method slots and their values from its parent NODES unit. That is, through inheritance, characteristics can be shared by all the children of NODE1 unit.

Through inheritance in KEE, slots and their values can both be inherited. The
### The NODES Unit in CSP Knowledge Base

<table>
<thead>
<tr>
<th>Unit: NODES in knowledge base CSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member of: CLASSES in GENERICUNITS</td>
</tr>
<tr>
<td>Members: NODE1, NODE2, NODE3, NODE4</td>
</tr>
</tbody>
</table>

**Member slot: ADD.VALUES from NODES**
- Inheritance: METHOD
- Valueclass: METHOD
- Comment: "Add one or more values to a node."
- Values:
  
  (lambda (self valuclist)
       (add.values self 'domain valuclist))

**Member slot: DELETE.VALUES from NODES**
- Inheritance: METHOD
- Valueclass: METHOD
- Comment: "Delete one or more values to a node."
- Values:
  
  (lambda (self valuclist)
       (remove.values self 'domain valuclist))

**Member slot: DENOTE from NODES**
- Inheritance: OVERRIDE.VALUES
- Valueclass: NUMBER
- Comment: "We denote variables by positive integer."
- Values: UNKNOWN

**Member slot: DOMAIN from NODES**
- Inheritance: OVERRIDE.VALUES
- Valueclass: UNKNOWN
- Comment: "Keep the domain values for a given node."
- Values: UNKNOWN

**Member slot: LIST.DOMAIN from NODES**
- Inheritance: METHOD
- Valueclass: METHOD
- Comment: "List all the domain values of a given node."
- Values:
  
  (lambda (self)
       (get.values self 'domain))

**Member slot: PREDICATE from NODES**
- Inheritance: METHOD
- Valueclass: METHOD
- Comment: "Unary predicate of a given node."
- Values: UNKNOWN

---

**Table 4.1: NODES unit and its internal structure.**
4.2. *CSP Representation in KEE*

member slots bequeath the slots to their descendants automatically. But the way slot values are inherited can be controlled by the inheritance role that is specified for a particular slot. The typical inheritance role is `override.values`. That is, if there is a local value in the slot, it is the value of the slot; otherwise, the values inherited from the parent are used. Although KEE provides a variety of inheritance mechanisms, we can describe an additional inheritance role of our own. Here we use inheritance only to `override.values`.

By using inheritance, we gain several advantages that make designing and modifying knowledge bases much easier. One advantage is that new units can be created easily. For instance, we can add a new node as a member of `NODES`. This new node automatically inherits its parent’s slots, such as `ADD.VALUES`, `DELETE.VALUES` and so on. It is only necessary to add new slot values that differentiate it from its parent. Another advantage is that modification is simplified. Any changes that are made in a class unit will be automatically inherited by all the descendants of this unit. For instance, if we change or modify the function of `ADD.VALUES` (the value of `ADD.VALUES` method slots) in `NODES` unit, the `ADD.VALUES` slot in all the members of `NODES` unit, such as `NODE1`, `NODE2`, and so forth, will be changed automatically.

2. Own slot:

An ‘own slot’ describes an attribute peculiar to a unit. That is, an own slot expresses a relationship involving its unit as an individual and no any interaction with other units. The `member` slots are inherited as `own` slots in the members of the class unit.

As shown in Table 4.2, all the slots, such as `ADD.VALUES` and so on, which inherited from `NODES`, become `own` slots.

We create `NODES` unit as a class unit. It contains six member slots including four method slots. These slots are `DOMAIN`, `PREDICATE`, `DENOTE`, `ADD.VALUES`, `DELETE.VALUES` and `LIST.DOMAIN`. The `DOMAIN` slot is used for storing a potential domain values of a node. The corresponding unary constraint (predicate)
### The NODE1 Unit in CSP Knowledge Base

<table>
<thead>
<tr>
<th>Unit: NODE1 in knowledge base CSP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Member of:</strong> NODES</td>
</tr>
<tr>
<td><strong>Own slot: ADD.VALUES from NODES</strong></td>
</tr>
<tr>
<td>Inheritance: METHOD</td>
</tr>
<tr>
<td>Valueclass: METHOD</td>
</tr>
<tr>
<td>Comment: “Add one or more values to a node.”</td>
</tr>
<tr>
<td>Values:</td>
</tr>
<tr>
<td>(lambda (self valuelist)</td>
</tr>
<tr>
<td>(add.values self 'domain valuelist))</td>
</tr>
<tr>
<td><strong>Own slot: DELETE.VALUES from NODES</strong></td>
</tr>
<tr>
<td>Inheritance: METHOD</td>
</tr>
<tr>
<td>Valueclass: METHOD</td>
</tr>
<tr>
<td>Comment: “Delete one or more values from a node.”</td>
</tr>
<tr>
<td>Values:</td>
</tr>
<tr>
<td>(lambda (self valuelist)</td>
</tr>
<tr>
<td>(remove.values self 'domain valuelist))</td>
</tr>
<tr>
<td><strong>Own slot: DENOTE from NODE1</strong></td>
</tr>
<tr>
<td>Inheritance: OVERRIDE.VALUES</td>
</tr>
<tr>
<td>Valueclass: NUMBER</td>
</tr>
<tr>
<td>Comment: “We denote the variable by positive integer.”</td>
</tr>
<tr>
<td>Values: 1</td>
</tr>
<tr>
<td><strong>Own slot: DOMAIN from NODE1</strong></td>
</tr>
<tr>
<td>Inheritance: OVERRIDE.VALUES</td>
</tr>
<tr>
<td>Valueclass: UNKNOWN</td>
</tr>
<tr>
<td>Comment: “Keep the domain values for a given node.”</td>
</tr>
<tr>
<td><strong>Own slot: LIST.DOMAIN from NODES</strong></td>
</tr>
<tr>
<td>Inheritance: METHOD</td>
</tr>
<tr>
<td>Valueclass: METHOD</td>
</tr>
<tr>
<td>Comment: “List all the domain values of a given node.”</td>
</tr>
<tr>
<td>Values:</td>
</tr>
<tr>
<td>(lambda (self)</td>
</tr>
<tr>
<td>(get.values self 'domain))</td>
</tr>
<tr>
<td><strong>Own slot: PREDICATE from NODE1</strong></td>
</tr>
<tr>
<td>Inheritance: METHOD</td>
</tr>
<tr>
<td>Valueclass: METHOD</td>
</tr>
<tr>
<td>Comment: “Unary predicate of a given node.”</td>
</tr>
<tr>
<td>Values:</td>
</tr>
<tr>
<td>(lambda (thisunit)</td>
</tr>
<tr>
<td>(ignore thisunit)</td>
</tr>
<tr>
<td>(if (= (length x) 5) T))</td>
</tr>
</tbody>
</table>

Table 4.2: NODE1 unit and its internal structure.
is kept in a PREDICATE slot. We denote nodes by positive integers which is put into DENOTE slot. This information is needed to realise arc consistent checking (further detail can be found in section 4.3.2). It is necessary for the end user to provide nodes and the associated domains of nodes and to supply the value for PREDICATE method slot (either as a lambda list or as a function name) when a node object is created. For instance, consider the PREDICATE method slot on the NODE1 unit. The value of the PREDICATE method slot is:

\[
\text{(lambda (thisunit x)} \\
\quad \text{(ignore thisunit)} \\
\quad \text{(if (= (length x) 5)} \\
\quad \quad \text{T)})
\]

The above unary constraint means that every value in this node's domain must have five letters. We can eliminate all the values from this node which do not satisfy this condition. Thus we can reduce the node's domain according to associated unary constraint. This operation is achieved in the NODE.CONSISTENCY method slot in the NODE.CONSISTENCIES unit (see section 4.3.1).

Each unary predicate must accept the following two parameters:

\[(\text{thisunit x)}\]

where thisunit is a required KEE method parameter and X is a variable parameter which can take any value of the domain associated with the created node unit. The PREDICATE method slot will return \text{true} after being executed if a value agrees with the associated unary predicate.

The ADD.VALUES member slot allows a user to explicitly add some values to a particular node's domain. DELETE.VALUES member slot can delete some values from corresponding domain according to the user's requirements. Among the six member slots, PREDICATE, ADD.VALUES and DELETE.VALUES are method slots (see Table 4.1). Method slots represent behavioural or procedural information. A unit can activate a method slot by sending it a message.

Each node in constraint network has its own domain, unary predicate and denotation. Thus, each node inherits these slots such as DOMAIN, PREDICATE, and so
forth from its parent unit NODES. But each node has its own properties and different values for these slots. So each node inherits its parent slots but not slot values. It is necessary for every node to have an ADD.VALUES and a DELETE.VALUES slot of its own. Therefore, each node receive these two slots and their values from NODES.

The syntax to use ADD.VALUES and DELETE.VALUES method is as follows:

```
(unitmsg < nodename > 'ADD.VALUES < valuelist >)
```

It is used for adding one or more values to a node.

```
(unitmsg < nodename > 'DELETE.VALUES < valuelist >)
```

It is used for removing one or more values from a node.

where < nodename > is the name you wish to give to the node object in a network and < valuelist > are the values which are to be added or removed.

LIST.DOMAIN method slot in NODES returns a list of the current values of the messaged node. Its messaging syntax is as follows:

```
(unitmsg < nodename > 'LIST.DOMAIN)
```

All the members of NODES inherit this method slot and its values.

We show two particular units, NODES and NODE1, and their internal structures implemented in KEE system (see Table 4.1 and Table 4.2). NODE1 is one of nodes in a constraint network. We can see clearly the relationship between class unit NODES and its member unit NODE1. The information about all the nodes in the network is kept in members of unit NODES, for example, NODE1, NODE2 and is specified when they are created.

In summary, NODES unit and its member units are used to keep all the information about nodes in a constraint network and to handle some basic node management.

### 4.2.2 ARCS Unit

The ARCS unit is a class unit which is used for storing information about arcs in the network. There are two member slots in an ARCS unit. One is the TWO.NODES slot which keeps the two nodes associated with the arc. The other is the BINARY-
4.2. CSP Representation in KEE

Figure 4.2: ARCS unit and its descendants.

<table>
<thead>
<tr>
<th>The ARCS Unit in CSP Knowledge Base</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unit:</strong> ARCS in knowledge base CSP</td>
</tr>
<tr>
<td><strong>Member of:</strong> CLASSES in GENERICUNITS</td>
</tr>
<tr>
<td><strong>Members:</strong> ARC1, ARC2, ...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Member slot: BINARY-PREDICATE from ARCS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inheritance:</strong> METHOD</td>
</tr>
<tr>
<td><strong>Valueclass:</strong> METHOD</td>
</tr>
</tbody>
</table>
| **Comment:** "Binary constraint of an arc."
| **Values:** UNKNOWN |

<table>
<thead>
<tr>
<th>The ARC1 Unit in CSP Knowledge Base</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unit:</strong> ARC1 in knowledge base CSP</td>
</tr>
<tr>
<td><strong>Member of:</strong> ARCS</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Own slot: BINARY-PREDICATE from ARC1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inheritance:</strong> METHOD</td>
</tr>
<tr>
<td><strong>Valueclass:</strong> METHOD</td>
</tr>
</tbody>
</table>
| **Comment:** "Binary constraint of an arc."
| **Values:** |
| (lambda (thisunit 11 12) |
| (ignore thisunit) |
| (if (char= (aref 11 2) |
| (aref 12 0)) |
| T)) |

<table>
<thead>
<tr>
<th>Own slot: TWO.NODES from ARC1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inheritance:</strong> OVERRIDE.VALUES</td>
</tr>
</tbody>
</table>
| **Comment:** "Keep two nodes connecting an arc."
| **Values:** |
| (NODE1 NODE2) |

Table 4.3: ARCS unit and one of its descendant ARC1 and part of their internal structure.
Chapter 4: Implementation of CSP Techniques in KEE

PREDICATE method slot which stores the corresponding binary constraint for an arc. BINARY-PREDICATE slot is either a lambda list or a function name which is defined by the user based on the specific problems. Each arc in the given constraint network is a member of an ARCS unit (see Figure 4.2). The internal representation of ARCS in KEE is shown in Table 4.3. Therefore, every arc inherits the slots in ARCS unit. Each arc consists of two nodes and a binary predicate based on the given network. The arc name is given by the user. It should be noted here that all the arcs in given constraint network are directional and connect two nodes. One node is the origin of the arc. The other node is the destination of the arc. The value of TWO.NODES slot is a list of associated two nodes (see Table 4.3). BINARY-PREDICATE contains a binary constraint between two nodes and should accept the following three parameters:

\[(\text{thisunit} \ I1 \ I2)\]

where \text{thisunit} is a required KEE method parameter. \(I1\) and \(I2\) are variables containing one of the values of the domain of the origin node and the destination node respectively. For a crossword puzzle, a specific binary predicate is represented as follows:

\[
(\lambda (\text{thisunit} \ I1 \ I2)
\begin{align*}
&\text{(ignore thisunit)}
&\text{(if (char= (aref I1 2) (aref I2 0))}
&\text{T})
\end{align*}
\]

Here a binary constraint arises when a word written across intersects a word down. The above constraint requires that the third letter of a word in the origin node be the same as the first letter of a word in the destination node.

It is worth noting that each arc, as implemented here, is uni-directional. When bi-directional arcs exist, it is necessary that two arcs (units) be created, one for each direction.
4.2. CSP Representation in KEE

![Diagram of PATHS unit and its descendants](image)

Figure 4.3: PATHS unit and its descendants.

The PATHS Unit in CSP Knowledge Base

<table>
<thead>
<tr>
<th>Method slot: NODELIST from PATHS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inheritance: OVERRIDE.VALUES</td>
</tr>
<tr>
<td>Comment: “Nodelist consists of all the nodes on the path.”</td>
</tr>
<tr>
<td>Values: UNKNOWN</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method slot: VALID.PAIRS from PATHS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inheritance: OVERRIDE.VALUES</td>
</tr>
<tr>
<td>Comment: “Valid.pairs keep all valid domain pairs discovered during path consistency checking”</td>
</tr>
<tr>
<td>Values: UNKNOWN</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Member slot: ADD.VALID.PAIRS from PATHS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inheritance: METHOD</td>
</tr>
<tr>
<td>Comment: “Add valid pairs to valid.pair slot.”</td>
</tr>
<tr>
<td>Values: CSP&gt;PATHS::ADD.VALID.PAIRS!method</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Member slot: DELETE.VALID.PAIRS from PATHS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inheritance: METHOD</td>
</tr>
<tr>
<td>Comment: “Delete valid pairs to valid.pair slot.”</td>
</tr>
<tr>
<td>Values: CSP&gt;PATHS::DELETE.VALID.PAIRS!method</td>
</tr>
</tbody>
</table>

Table 4.4: PATHS unit with one of its descendants, and part of their internal structure.
4.2.3 PATHS unit

The PATHS unit is used for storing information about path and valid pairs. All the paths in a given constraint network are the members of PATHS unit. The PATHS unit and its members are shown in Figure 4.3. The members of PATHS unit, that is, all the paths in the network, are specified by the user. It contains four member slots. The NODELIST slot keeps all the names of nodes along the path. The VALID.PAIRS slot preserves all valid domain value pairs discovered during path consistency checking. ADD.VALID.PAIRS and DELETE.VALID.PAIRS method slots are used to add or delete valid pairs to or from paths. The internal structure of PATHS is given in Table 4.4. It is necessary for the user to provide path names and a corresponding nodelist. The VALID.PAIRS slot is empty before performing path consistency checking. No predicates need to be defined for path consistency checking since it utilizes the binary predicates found within the associated arc units.

4.2.4 BOSS unit

The BOSS unit is built to deal with interface and management. We make use of this unit to input nodes, domains, arcs, paths, unary and binary predicates. It contains several own method slots which specify an attribute particular BOSS unit, including INPUT.NODES.AND.UNARY.PREDICATES, INPUT.ARCS.AND.BINARY.PREDICATES, INPUT.PATHS, MAKE.LINKS and OUTPUT.DOMAIN etc. (see Table 4.5). We will give more detailed information about these own slots:

INPUT.NODES.AND.UNARY.PREDICATES slot allows the user to input node names which are named by the user to the nodes of a constraint network, the associated domains and unary constraints. This slot organizes all the nodes which are given by the user as members (descendants) of a NODES unit. These nodes inherit slots and/or slot values from a NODES unit and have their own values, such as domains and unary predicates. The user can follow the instruction on the screen to input corresponding knowledge. The syntax for invoking INPUT.NODES.AND.UNARY.PREDICATES method slot is as follows:

(unitmsg 'BOSS 'INPUT.NODES.AND.UNARY.PREDICATES)
4.2. CSP Representation in KEE

The BOSS Unit in CSP Knowledge Base

<table>
<thead>
<tr>
<th>Unit: BOSS in knowledge base CSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member of: CLASSES in GENERICUNITS</td>
</tr>
</tbody>
</table>

Own slot: INPUT.NODES.AND.UNARY.PREDICATES from BOSS

- Inheritance: METHOD
- Valueclass: METHOD
- Comment: "Input nodes and unary predicates."
- Values:
  - CSP>BOSS::INPUT.NODES.AND.UNARY.PREDICATES

Own slot: INPUT.ARCS.AND.BINARY.PREDICATES from BOSS

- Inheritance: METHOD
- Valueclass: METHOD
- Comment: "Input arcs and binary predicates."
- Values:
  - CSP>BOSS::INPUT.ARCS.AND.BINARY.PREDICATES

Own slot: MAKE.LINKS from BOSS

- Inheritance: METHOD
- Valueclass: METHOD
- Comment: "Set up array *link* according to input information.
  ARC(CONSISTENCY2 needs to use this array."
- Values:
  - CSP>BOSS::MAKE.LINKS

Own slot: OUTPUT.DOMAIN from BOSS

- Inheritance: METHOD
- Valueclass: METHOD
- Comment: "Display all the nodes domains in constraint network."
- Values:
  - CSP>BOSS::OUTPUT.DOMAIN

Table 4.5: BOSS unit and part of its internal structure.

It is necessary for the user to supply the unary constraint. An example of the format for inputing unary predicate is:

(lambda (thisunit x)
  (ignore thisunit)
  (programbody) )

where x is a variable parameter which takes one member of a node domain. Program body is defined by the user according to the related unary constraint. The above constraint returns true for each member of domain if the member satisfies the corresponding constraint. Otherwise it returns false. We make use of this result to achieve node consistency checking.

INPUT.ARCS.AND.BINARY.PREDICATES slot asks the user for input to all the arcs in a constraint network, the origin and destination nodes of the arcs and the binary constraints between these nodes. This slot organises all the arcs which are
given by the user as members (descendants) of the ARCS unit. These arcs inherit some slots and/or slot values from NODES unit. These arcs in the network have their own values of predicates. The INPUT.ARCS.AND.BINARY.PREDICATES prompts the user to input arcs into the network and the corresponding binary predicates. The syntax of invoking this message is as follows:

```
(unitmsg 'BOSS 'INPUT.ARCS.AND.BINARY.PREDICATES)
```

The binary predicate format is:

```
(lambda (thisunit l1 l2)
    (ignore thisunit)
    (programbody)
)
```

where l1 and l2 are variables standing for a member of origin and destination node domains respectively. It is necessary for the user to define the “programbody” corresponding to a binary predicate. It returns true for a pair of elements between the origin and the destination node domains if they satisfy the binary constraint. This result is utilized when performing arc consistency checking.

We use INPUT.PATHS method slot to input paths in a network. The nodelist along path and the path name is provided by the user.

The OUTPUT.DOMAIN slot can display all the nodes’ domains in one output window. We can use the following syntax to invoke display domains:

```
(unitmsg 'BOSS 'OUTPUT.DOMAIN)
```

We have already demonstrated how to represent nodes, arcs, paths of a constraint network built in the NAPCC module. We have also shown the syntax of unary and binary predicates and how to invoke associated messages. Now let us consider how to employ KEE to realize node, arc, and path consistency checking (the main parts of NAPCC module). Thus, a constraint network can become a consistent network. A number of domain values which can never be part of any possible global solution can be eliminated.
4.3 Consistency Checking Implementation in KEE

NAPCC module includes the NAP.CONSISTENCIES class unit (as shown in Figure 4.4). The NAP.CONSISTENCIES unit contains all the algorithms used in consistency checking such as node, arc, and path consistency checking. It has three member units, each performing a particular consistency checking. These units are NODE.CONSISTENCIES, ARC.CONSISTENCIES and PATH.CONSISTENCIES, which are members of NAP.CONSISTENCIES. We will discuss each of these in the following subsection.

4.3.1 NODE.CONSISTENCIES Unit

The NODE.CONSISTENCIES unit handles node consistency checking. It contains three method slots. These slots are NC1, NODE.CONSISTENCY and NODE.CONSISTENCY.FOR.SOME.NODES (as shown in Table 4.6). Each node can be made consistent by performing the domain reduction operation:

$$D_i \leftarrow D_i \cap \{x | P_i(x)\}$$

This is implemented by NC1 method slot. In Table 4.6 the algorithm for one node consistency checking is presented. The sending message syntax for invoking one node consistency checking is as follows:

(unitmsg 'NODE.CONSISTENCIES 'NC1 < nodename >)

The aim of sending the above message is to remove values from the node's domain which do not satisfy the associated unary predicate of the node. We should perform node consistency checking on all the nodes in the constraint network in order to
Table 4.6: NODE.CONSISTENCIES unit and its internal structure.

make a network node consistent. We give the following definition of node consistent network:

**Definition 4.5** A constraint network is node consistent if and only if all its nodes are consistent.

A given constraint network for a specific problem can be made node consistent in a single pass over the nodes. The NODE.CONSISTENCY method slot is primarily designed to achieve this (Table 4.7). To send a message to the NODE.CONSISTENCIES method slot in the NODE.CONSISTENCIES unit, we use:

```
(unitmsg 'NODE.CONSISTENCIES 'NODE.CONSISTENCY)
```

The given constraint network can be made node consistent after executing above command.

### 4.3.2 ARC.CONSISTENCIES Unit

Arc consistency checking, as implemented here, is based on the information that arcs between two nodes can be shown to be consistent on the basis of an associated
4.3. Consistency Checking Implementation in KEE

The **NODE.CONSISTENCY** Slot of the **NODE.CONSISTENCIES** Unit

<table>
<thead>
<tr>
<th>Own slot: NC1 from <strong>NODE.CONSISTENCIES</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Inheritance: <strong>METHOD</strong></td>
</tr>
<tr>
<td>Valueclass: <strong>METHOD</strong></td>
</tr>
<tr>
<td>Comment: &quot;Make all the nodes consistent.&quot;</td>
</tr>
<tr>
<td>Values:</td>
</tr>
<tr>
<td>(lambda (thisunit node)</td>
</tr>
<tr>
<td>(ignore thisunit)</td>
</tr>
<tr>
<td>(let ((nodes (unit .descendants 'node 'member)))</td>
</tr>
<tr>
<td>(dolist (node nodes)</td>
</tr>
<tr>
<td>(unitmsg 'node.consistencies</td>
</tr>
<tr>
<td>'nc1</td>
</tr>
<tr>
<td>node))))</td>
</tr>
</tbody>
</table>

Table 4.7: **NODE.CONSISTENCY** unit and its internal structure.

The **ARC.CONSISTENCIES** Unit in CSP Knowledge Base

<table>
<thead>
<tr>
<th>Unit: <strong>ARC.CONSISTENCIES</strong> in knowledge base CSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member of: <strong>NAP.CONSISTENCIES</strong></td>
</tr>
<tr>
<td>Own slot: REVISE1 from <strong>ARC.CONSISTENCIES</strong></td>
</tr>
<tr>
<td>Inheritance: <strong>METHOD</strong></td>
</tr>
<tr>
<td>Valueclass: <strong>METHOD</strong></td>
</tr>
<tr>
<td>Comment: &quot;Using arc as an argument, make it consistent.&quot;</td>
</tr>
<tr>
<td>Values:</td>
</tr>
<tr>
<td>CSP&gt;ARC.CONSISTENCIES::REVISE1!method</td>
</tr>
</tbody>
</table>

Own slot: **ARC.CONSISTENCY1** from **ARC.CONSISTENCIES**

| Inheritance: **METHOD**                           |
| Valueclass: **METHOD**                           |
| Comment: "One simple approach to make all the arcs consistent but it is inefficient." |
| Values:                                            |
| CSP>ARC.CONSISTENCIES::ARC.CONSISTENCY1!method    |

Own slot: REVISE2 from **ARC.CONSISTENCIES**

| Inheritance: **METHOD**                           |
| Valueclass: **METHOD**                           |
| Comment: "Using two nodes as arguments, make the arc between them consistent." |
| Values:                                            |
| CSP>ARC.CONSISTENCIES::REVISE2!method             |

Own slot: **ARC.CONSISTENCY2** from **ARC.CONSISTENCIES**

| Inheritance: **METHOD**                           |
| Valueclass: **METHOD**                           |
| Comment: "A better approach to make the constraint network arc consistent." |
| Values:                                            |
| CSP>ARC.CONSISTENCIES::ARC.CONSISTENCY2!method    |

Own slot: **ARC.CONSISTENCY.FOR.SOME.ARCS** from **ARC.CONSISTENCIES**

| Inheritance: **METHOD**                           |
| Valueclass: **METHOD**                           |
| Comment: "Make constraint network arc consistent after changing some information about arcs." |
| Values:                                            |
| CSP>ARC.CONSISTENCIES::ARC.CONSISTENCY.FOR.SOME.ARCS!method |

Table 4.8: **ARC.CONSISTENCIES** unit and its internal structure.
binary predicate, provided that each node associated with the arc has already been shown to be node consistent. Thus it is vital that all the necessary node consistency checking be carried out prior to arc consistency checking.

For each arc we use following arc consistency domain restriction operation:

\[ D_i \leftarrow D_i \cap \{x | \exists y (y \in D_j) \land P_{ij}(x, y)\} \]

that is, for every element in \( D_i \) there is at least one element in \( D_j \) such that they satisfy the constraining binary predicate. Arc \( (i, j) \) between \( D_i \) and \( D_j \) can be made arc consistent by removing from \( D_i \) all elements that have no corresponding element in \( D_j \) satisfying the associated binary constraint. The above operation is implemented by REVISE1 method slot in ARC.CONSISTENCIES unit (see Table 4.8).

To perform an arc consistency check, the following message syntax should be used:

(unitmsg < arcname > 'revise1)

The REVISE1 method slot will invoke the binary constraint related to the arc for each value of the origin node domain of the arc. Any value of the origin node domain for which the binary constraint fails will be removed from the domain.

We perform arc consistency checking on all the arcs in the constraint network in order to make it arc consistent. We give the following definition of an arc consistent network:

**Definition 4.6** A constraint network is arc consistent if and only if all its arcs are consistent.

It should be noted that a single pass of the arc consistency operation over all the arcs will not guarantee that the given network is arc consistent. For instance, arc \( (i, j) \) must be consistent after immediately applying (unitmsg arcij 'revise1) to arc \( (i, j) \). However, it may not remain arc consistent because members in \( D_j \) may subsequently be removed by applying (unitmsg arckj 'revise1) to some arc \( (j, k) \). Therefore, a single pass through all the arcs applying (unitmsg arc 'revise1) to each arc, is not sufficient. The simplest method for achieving an arc consistent network is to repeat such a pass until there is no change reduction in any domain in a complete pass.

The above method achieves network arc consistency via the following syntax:
4.3. Consistency Checking Implementation in KEE

This method is inefficient, because a successful deletion of an arc on a particular pass causes all the arcs to be checked on the next pass whereas in fact only a small fraction of them could possibly be affected.

In order to avoid this poor behavior, we adopt the following steps:

1. Push all the arcs in constraint network into a queue Q;
2. Pop one arc \((i, j)\) from the queue Q and apply an arc consistency check to it;
3. If successful revision (reduction of one or more values from the node domain) on the arc \((i, j)\), then the only additional arcs that need to be reconsidered are all those that lead to \(i\), \((p, i)\), with the exception of \((j,i)\) because it cannot have become inconsistent as a direct result of the deletions made in \(D_i\) by applying revision to arc \((i, j)\). Then push only those arcs \((p, i)\) to the queue Q which have not appeared in Q.
4. Then go to step 2 until the queue Q is empty.

In order to implement step 3, we need to get all the arcs which lead to node i. We use a two-dimensional array to store the connection of all the nodes. The index of array is the same as the value of DENOTE in a node. If there exists a link (an arc) between i and j, the corresponding position in the array \((i,j)\) is 1. Otherwise it is zero. That is why we denote a variable by a positive integer and keep the number in the DENOTE slot in each node. The data of this array will be automatically created according to certain information about nodes and arcs in our system.

The ARC.CONSISTENCY2 method slot in the ARC.CONSISTENCIES unit takes advantage of the above strategy. This new procedure is more efficient than the simple method. In using this arc consistency check, the following message syntax should be used:

\((\text{unitmsg 'ARC.CONSISTENCIES 'ARC.CONSISTENCY2})\)

The ARC.CONSISTENCIES unit also contains a REVISE2 method slot, which it is primarily designed for use by the ARC.CONSISTENCY2 method. Its function
Chapter 4: Implementation of CSP Techniques in KEE

is nearly the same as REVISE1. The only difference is that REVISE1 needs arc as parameter while REVISE2 needs two nodes as parameters.

4.3.3 PATH.CONSISTENCIES Unit

Path consistency checking in this module is handled by the PATH.CONSISTENCIES unit. In an analogous manner to the ARC.CONSISTENCIES unit, the PATH.CONSISTENCIES unit assumes that all arcs in a particular path are arc consistent prior to path consistency checking. For each path we use the following path consistency domain restriction operation:

\[(D_{i_0}, D_{i_m}) \leftarrow (D_{i_0}, D_{i_m}) \cap \{(x, y) | (\exists z_1) \ldots (\exists z_{m-1})
\wedge P_{i_0 i_1}(x, z_1) \wedge P_{i_1 i_2}(z_1, z_2) \wedge \ldots \wedge P_{i_{m-1} i_m}(z_{m-1}, y)\}\]

That is, for every pair in \(D_{i_0}\) and \(D_{i_m}\) there is at least one series element in \(D_{i_1}, \ldots, D_{i_{m-1}}\) such that the pair of elements satisfy the corresponding binary predicate. A path can be made path consistent by removing all the pair elements between \(D_{i_0}\) and \(D_{i_m}\) that have no corresponding series elements in \(D_{i_1}, \ldots, D_{i_{m-1}}\) satisfying the associated binary constraint. The above operation is achieved by ONE.PATH.CONSISTENCY method slot in PATH.CONSISTENCIES unit. To achieve one path consistency checking, the following message syntax is used:

```
(unitmsg 'PATH.CONSISTENCIES 'ONE.PATH.CONSISTENCY PATH)
```

where path is the name of a path which needed to perform path consistency checking. No predicates need to be defined for path consistency checking since it uses the predicates already used for the path’s constituent arcs. Path of length \(m\) has only one starting node domain \(D_{i_0}\) and one ending node \(D_{i_m}\). A path consistency check whether each pair between \(D_{i_0}\) and \(D_{i_m}\) is a valid pair or not. We store all the valid pairs in a VALID.PAIRS data slot on a path unit discovered during path consistency checking. This slot can be used explicitly during a solution search to reduce search space.

To make all the paths in a constraint network consistent, we use PATH.CONSISTENCY method slot in PATH.CONSISTENCIES units. The syntax of sending a message to perform path consistency checking is as follows:
4.3. Consistency Checking Implementation in KEE

To achieve path consistency, the user has to provide all the paths of a given constraint network. The system is not set up to get all the paths automatically. In the cases studies, we only employ node and arc consistency checking for pre­processing. So an example is given just to show how to improve search efficiency by performing node and arc consistency checking.

4.3.4 An Example

A simple application of using CSP techniques is the crossword puzzle problem [Mackworth 87]. Here we use a crossword puzzle problem as an example to demonstrate how the search space is greatly reduced after applying the NAPCC module built in to KEE. We consider the puzzle in Figure 4.5 and use words in the given word list. Eight words are required to fill in the gaps in Figure 4.5. This can be formulated as a CSP problem by creating a variable for each gap to be filled and associating with each variable the word list as the domain. The unary constraint set \{ \Pi \} specifies the word length. For instance, \Pi_1 requires that the word in variable \( v_1 \) starting at 1 across have five letters. The binary constraint set \{ \Pi_{ij} \} arises when a word across intersects a word down. For example, the constraint between variable \( v_1 \) and \( v_2 \), namely \Pi_{12} requires that the third letter of \( v_1 \) be the same as the first letter of \( v_2 \). The constraint network for the crossword puzzle is shown in Figure 4.6 (a). The initial domain for each variable is inside the vertex for that variable.

If we use the generate-and-test approach to solve this problem, the number of
Chapter 4: Implementation of CSP Techniques in KEE

Figure 4.6: (a). The original constraint network for the crossword puzzle. (b). The constraint network after applying the NAPCC module for node consistency checking.

Figure 4.7: (a). The detailed process of performing arc consistency checking. (b). The constraint network after applying the NAPCC module for arc consistency checking.
4.3. Consistency Checking Implementation in KEE

Different assignments to be tested is

$$\prod_{i=1}^{n} N_i$$

where $N_i$ is the number of words in each variable domain and $n$ is the number of nodes. Here $N$ equal to 14 and $n$ is 8. Thus

$$\prod_{i=1}^{n} N_i = 14^8$$

It is better to use node consistency checking in the NAPCC module built into KEE before performing searching solutions. We employ node consistency checking to eliminate those words which do not satisfy the corresponding unary word length constraint once and for all. Thus the domain of each variable is reduced (in Figure 4.6 (b)).

If we still use the generate-and-test method on the node consistent network to find solutions, the number of different assignments to be tested now is

$$\prod_{i=1}^{n} N_i = 5^8$$

Thus the search space is greatly reduced.

We can work further on the constraint network via utilizing NAPCC module. We apply arc consistency checking on the revised constraint network by ARC.CONSISTENCY2. The arcs to be examined are put on a queue $Q$ in the order (1 2), (1 3), (2 1), ..., (8 6). When words are eliminated from the domain of a node, all the arcs into that node which currently are not waiting on the queue (except the reverse of the revised network).
the arc causing the deletion) are added to the queue. In Figure 4.7 (a) the numbers following the deleted words give the order in which they are deleted. The result of applying ARC.CONSISTENCY2 to the Figure 4.6 (b) is shown in Figure 4.7 (b). Each domain is reduced to only one value. There exists the unique solution shown in Figure 4.8 to this crossword puzzle problem.

This example demonstrates how the search space is greatly reduced by applying the NAPCC module for consistency checking. It shows that the efficiency of the search for a solution can be improved significantly.

### 4.4 Discussion

Although KEE is a very powerful tool for solving AI problems, it doesn’t provide an environment for implementing CSP techniques. Thus, it is inconvenient for the KEE to establish expert systems where CSP techniques are required. As we will find in our case studies in Chapters 6 and 7, the CSP techniques are very useful for the improvement of searching efficiency in non-CSP experts systems such as systems built on KEE.

Considering this situation, we have explained how to represent node, arc and path and relevant knowledge for CSP in KEE. We have built and implemented a node-arc-path consistency checking (NAPCC) module to realize the basic CSP techniques. Hence the CSP techniques of node, arc and path consistency checking, are made available for establishing non-CSP expert systems, e.g. rule-based systems on KEE. Finally we have shown, by an example, how to use the NAPCC, and discussed the applications of the NAPCC. We will demonstrate in the case studies the usefulness of the CSP techniques in expert systems on KEE.
Chapter 5

Two Strategies for User-Directed Search

5.1 Introduction

We have discussed CSP techniques and developed CSP module in Chapters 3 and 4, which play an important role in terms of search efficiency in User-Directed Search (UDS) studies. CSP module is used in performing data process before starting searching for a possible solution in achieving UDS.

Often, a number of possible solutions exist for a given practical problem. In applications we are concerned with, problem solving systems are usually used for finding only one acceptable solution to the given problem, while acceptance or rejection of this solution will be left to the user. There are various ways to find a possible solution. Generally, it is not known which branch of a search tree leads to an acceptable solution. In order to search for a solution which satisfies the user, we have developed an environment which allows the user to interact with the whole process of searching. In this way, the process of problem solving can be interrupted by the user and can accept advice from the user. Working in this environment, the user can interact with the solution searching and guide it in his most preferred directions, so that the user-desired solution may be found in the most efficient way.

In order to establish such an environment, we introduce a search scheme which
we call UDS. The work of UDS was motivated by the consideration that the user's involvement in problem solving may improve efficiency. In this scheme, the problem solver offers a possible solution which satisfies the initial set of constraints. If the user is not satisfied with the solution for some reasons, how can the problem solver make effective use of the results obtained for further searching and be guided by the user? UDS is introduced to answer this question – it may become a useful part of many practical expert systems.

The UDS scheme enables the user to control and direct the search during the course of problem solving. In the next section, we discuss a general UDS. We then present two strategies for implementing UDS. One strategy, backjumping, utilizes TMS and KEEworlds in KEE[Intellicorp 87] to achieve it. Using this strategy, the system can transform the requirements of the user into deduction rules which cause the creation of justifications and can be used to find the decision point which conflicts with the user's requirements in the search tree. Finding such a point allows back up to and continuation of further search from that point. The method can avoid a number of unnecessary backtracks.

The other strategy is based on effective transfer of information from one solution to the next solution. In this approach, we have classified the variables into four different groups. These are named hold, base, free and change groups respectively. This classification can make UDS more flexible, and therefore UDS can provide a very useful environment where the suitable solution searching can be conveniently monitored and directed.

We describe here the basic algorithms of implementing UDS. In the following chapters, we will use UDS schemes in two practical cases — a truck dispatching problem and a crossword puzzle problem.

5.2 UDS

In practice, there may exist a number of solutions satisfying a set of constraints in a given problem. The search space of all the solutions can be very large. However, only an acceptable solutions which satisfy the user are required in many practical
5.3. Backjumping

applications. Thus in these cases we need not waste time in exploring additional solutions if we have found an acceptable solution to the problem under consideration. The user may have a preference in accepting one solution while declining another. Hence, it is possible that the system searches in the wrong direction so that the solution does not satisfy the user. We introduce user-directed search (UDS) to such cases so that the user can intervene and direct the search process. In these cases whether a solution is satisfactory or not is determined by the user. UDS can provide a very useful environment in which the solution search can be conveniently monitored and directed. Using UDS, we can interrupt the search procedure according to the solutions obtained, by changing requirements or constraints and by keeping some results obtained while discarding others in further searching.

In general, a complete UDS should include two parts:

1. Constraint Strengthening;

2. Constraint Weakening.

Constraints are strengthened when further constraints or restrictions are added by the user during the course of solution searching. Constraint strengthening can be used to guide and control the search to find a solution which satisfies the user.

Constraints are weakened when one or more constraints or restrictions are relaxed. Sometime, a problem is over-constrained and no solution can be found. Thus, the initial constraint set has to be relaxed in order to obtain an acceptable solution.

In this thesis, we concentrate on problem involving constraint strengthening. Two strategies for achieving UDS are based on constraint strengthening via additional conditions according to the user's requirements.

5.3 Backjumping

A possible solution can be obtained with a set of initial constraints. However, the user may not be satisfied with it because it is not the solution which is expected by the user or some particular circumstances occur. For instance, driver Brown is supposed to driver truck T1 from the solution S1. But the user prefers that driver
Brown drives truck T2. Thus, another possible solution, which has to satisfy the set of initial constraints plus additional conditions required by the user, needs to be found. Generally, finding and recording all the solutions for the given problem may be too costly and may not be efficiently used while only one feasible solution is required and when more constraints are added by the user according to actual situation. Obviously, it is inefficient if we start the search from the very beginning when minor changes to the initial constraints are introduced. One problem of concern is how to effectively make use of the information of the previous search to find a new solution which satisfies the newly-changed constraint set.

If the user is not satisfied with the solution obtained, a new solution has to be sought to suit the user's new conditions. The system then transforms the conditions into deduction rules. The premises of the deduction rules are the negative of the new conditions, while the conclusion of deduction rules is \textit{false}. In the truck dispatching problem, for instance, the user requires that the driver Brown drives the particular truck T2, but driver Brown is already assigned to drive truck T1 in the solution obtained. This is a new condition in addition to the static constraints. We then transform this new constraint into the following deduction rule:

\begin{verbatim}
while
    (the driver of ?trip is Brown)
    (the truck of ?trip is ?t)
    (not (equal ?t T2))
believe
    false
\end{verbatim}

This kind of rule is created as a member of the class of \textit{user.constraint.rules} which can be used to find the choice contributing to the contradiction. We have implemented the UDS as a specialized problem solver integrated with an assumption-based truth maintenance system (ATMS), KEEworlds and rule system in KEE [Intelllicorp 87].

To introduce our UDS algorithm, we first give a description of the problem we are attempting to solve. We assume that there are \( n \) variables, \( v_1, \ldots, v_n \). Each variable
5.3. Backjumping

Global variables:

\( C \) is a set of constraints. \( \Gamma : \) a set of new user’s requirements. \( v_1, v_2, \ldots, v_n \) are variables of the problem. \( j \) is a variable. \( D_1, D_2, \ldots, D_n \) are domains of \( v_1, v_2, \ldots, v_n \) respectively. solution_flag is a variable which is ‘yes’ if a possible solution is obtained, otherwise it is ‘no’. satisfy_flag is a variable which is ‘yes’ if a solution satisfies the user, otherwise it is ‘no’.

(a) Global Variables

```
procedure search1(i, v_i)
begin
1: if i > n then
   begin
      output(v_1, v_2, \ldots, v_n)
      solution_flag <- ‘yes’ ;a possible solution is found
      call UDS1
      if (satisfy_flag = ‘yes’) or (solution_flag = ‘no’)
      then exit else i = j + 1
   end
else if i = 0 then ;no solution exists
   begin
      solution_flag <- ‘no’
      print(“No solution exists”)
      exit
   end
2: for d = each element of \( D_i \) do
   begin
      v_i <- d
      conflict_flag <- check(i, v_i, C); invoke constraint.rules to check
      whether \( v_i \) is consistent assignment, return ‘no’ if it doesn’t conflict with
      constraint sets, otherwise return ‘yes’
      if conflict_flag = ‘no’
      then search(i + 1, v_{i+1})
   end
end
```

Figure 5.1: The algorithm of searching for a possible solution.
Chapter 5: Two Strategies for User-Directed Search

$v_i$ has a set of possible values, $D_i$, called the variable domain. The problem we consider is to find a solution $s$ which is an assignment of values to the variables and satisfies a set of constraints $C$. The algorithm of searching for one possible solution and calling UDS1 is given in Figure 5.1. Figure 5.1(a) shows the global variables which are shared by the algorithms.

The procedure `search1` in Figure 5.1(b) is a description of the search algorithm. $i$ is an integer representing the fact that the $i$th variable will be assigned and will increase with each variable assignment during the search. It takes on the value 1 at the initial call. The search traverses the variables in a pre-determined order. We use rule system, KEEworlds, ATMS in KEE to achieve searching, checking constraints, and recording the intermediate results during the search (For further details refer to Chapters 6 and 7). We use worlds to represent each state of the search. The search is initiated by calling `search` with $i = 1$. When a possible solution is obtained, the UDS1 (Figure 5.2) is invoked. Before we call procedure `search(1, v_i)`, we use CSP techniques to reduce variable domains. This is to eliminate domain values which will not appear in any possible solution satisfying the constraints $C$. The detailed description of CSP techniques has been presented in Chapters 3 and 4.

Figure 5.2 shows the main control process and UDS1. The user can decide whether the solution obtained is acceptable or not. If not, another solution should be sought. If the user adds new requirements, the next possible solution must also satisfy the new conditions from the user. Line 2 in UDS1 will achieve backjumping to the right point to start further searching. The backjumping algorithm is shown in Figure 5.4. In the UDS1, new requirements from the user can be given in the form of constraints. We transfer such constraints into deduction rules to create ATMS justifications. These justifications will be used to represent generalized dependencies and thus to control search process. Thus we can find the right point to start a further search based on the user’s requirements.

In our truck dispatching system, for example, a new requirement is given by the user when a solution is found and the user is not satisfied with it. The requirement is a constraint that driver White can use any truck except the particular truck T4.
5.3. Backjumping

Main()

begin
1: order(v1, v2, ..., vn) ; the variables are instantiated
   in the order v1, v2, ..., vn
2: search(1, v1) ; search for a solution
3: if satisfy_flag = 'yes'
   then
      begin
      print("A satisfactory solution is obtained")
      exit ; exit from search1
      end
   else exit ; no solution exists
end

procedure UDS1

begin
1: input(satisfy_flag) ; ask the user whether is satisfied
   with the solution
2: if solution_flag = 'no' then
   begin
      col-reqs
      j ← backjump(Γ)
      ; collect user's new requirements
      ; invoke user.constraint.rules, find the
      ; choice for vj which does not conflict with
      ; the user's requirements.
      ; prepare for searching for another solution
      ; according to the user's requirements.
      ; this resets control to the jth level of
      ; the current search stack.
   end
end

Figure 5.2: Main control algorithm and UDS1.
procedure col-reqs
\( \delta : \) a new requirement
begin
1: \( \Gamma \leftarrow \{\}\)<br>
2: flag \( \leftarrow \) enquire(more) ; whether to input new requirement
3: while flag = ‘yes’ do
\begin{align*}
3.1: & \quad \delta \leftarrow \text{input(requirement)} \\
3.2: & \quad \text{transform}(\delta) \quad \text{; transform user's requirement into} \\
3.3: & \quad \Gamma \leftarrow \Gamma \cup \{\delta\} \quad \text{; deduction rule which is a member of} \\
3.4: & \quad \text{flag} \leftarrow \text{enquire(more)} \quad \text{; whether input more requirement}
\end{align*}
end
4: \( C \leftarrow C \cup \Gamma \)\end{enumerate}

Figure 5.3: An algorithm for collecting user's new requirements.

backjump(R)
R: a set of the user's requirements
begin
\begin{align*}
\text{for } i = n \text{ to } 1 \text{ do} \\
\text{begin} \\
\text{check\_flag} \leftarrow \text{check}(i, v_i, R) \quad \text{; check whether it conflicts with the} \\
\text{requirements } R, \text{ if it does, make the} \\
\text{choice nogood and return('yes')} \\
\text{if it does not conflict with } R, \text{ return('no')} \\
\text{if check\_flag = 'no' then return}(i) \quad \text{end} \\
\text{end} \quad \text{return}(0)
\end{align*}

Figure 5.4: Backjumping algorithm of UDS1.
5.4. Constraint Recording

The requirement can be transformed by our system into the following deduction rule:

\[
\text{while} \quad \begin{align*}
&\text{(the driver of \(?trip\) is White)} \\
&\text{(the truck of \(?trip\) is T4)}
\end{align*}
\text{believe} \quad false
\]

In practical cases, deduction rules like this will be invoked with the highest privilege and, accordingly, ATMS justifications will be created. These ITMS justifications can be used to find how the previous solution conflicts with the requirements. We use worlds to represent the decision points in the search tree. If the facts of a world satisfy the conditions of deduction rules, that means that the constraints are not satisfied and the conclusion false is added to the world, which then becomes inconsistent. The system will avoid those worlds which are inconsistent from further consideration. So each world will then be checked back along the search tree to find whether it conflicts with the new requirements. If it does, it will become nogood and all the possibilities for this choice will be ignored. Checking will go to higher worlds until a world does not conflict with the user's requests. We utilize KEEworlds, Rulesystem, and TMS in KEE to implement this. Hence, we can invoke user.constraint.rules which consists of deduction rules to find the right point (world) for further search using a large constraint set (an initial constraint set plus the new requirements from the user). By this process, we can avoid a number of unnecessary backtracks.

5.4 Constraint Recording

During the course of solution searching, some choices or decisions lead to the dead-ends. In order to improve search efficiency, we require that the same dead-end will not arise again in the continuation of the search. With this aim, we have developed constraint recording in order to remember the reasons for the dead-end so that the same mistake (useless search) will be avoided in further search.
Normally, we use false as the conclusion of the deduction rules which are important for expressing constraints. However, when using only deduction rules for checking constraints, there exists the problem that the same wrong choice that leads to inconsistent may be arise again during the course of searching. Hence, we create some slots (a detailed description of slots can be found in Chapter 2) to store the information about dead ends. Furthermore, we take more action in the conclusions of the constraint rules so that we can remember the information which leads to the dead ends. In the truck dispatching problem, for instance, the requirement that the driver and the truck must be in the same place can be expressed as follows:

if
(?t is in class Trips)
(the location of (the driver of ?t) is ?d1)
(the location of (the truck of ?t) is ?v1)
(not (equal ?d1 ?v1))
then
(lisp (add.value (the truck of ?t)
               'truck.nogood.driver
               (the driver of ?t)))
(lisp (add.value ?t
               'trip.nogood.driver
               (the driver of ?t)))
(lisp (assert 'false nil world))

How to achieve constraint recording will be described in detail in Chapter 6.

5.5 Variable Classification

Since only one acceptable solution is required while many solutions exist in many practical applications, it is usual that the user may show his preference in accepting one solution while declining others. For example, a possible solution may be found, but the user can find it unsatisfactory because it differs from what the user had
5.5. Variable Classification

```
Main()
begin
1: order(v1, v2, ..., vn) ; the variables are instantiated
   in the order v1, v2, ..., vn
2: search2(1, v1) ; search for a solution
3: if solution_flag = 'yes' then ; a possible solution is found
   begin
      3.1: input(satisfy_flag) ; ask the user whether is satisfied
           with the solution
      3.2: while (solution_flag = 'yes') and (satisfy_flag = 'no') do
           begin
              UDS2
              if solution_flag = 'yes'
                 then input(satisfy_flag)
           end
      3.3: if satisfy_flag = 'yes'
           then return('success') ; the user accepts the results
           ; return with the current solution
           end
4: return('no solution') ; no solution exists
end

```

Figure 5.5: Main control Algorithm for UDS2.

expected or it conflicts with some conditions which had not considered beforehand. In searching for another solution, efficiency can be gained by taking advantage of the solution obtained. In these cases, there is a similarity between the solution obtained (even though it is considered as unacceptable or incomplete) and solution to be obtained. The achievement of maximum possible efficiency is largely dependent on making effective use of the solution obtained during the course of the further search. In other words, maximum information should be transferred from one solution to another solution.

We have created a very useful interface for UDS to make the procedure of solution searching more user-friendly, efficient, and controllable. With this interface, we can conveniently and efficiently direct the whole process of searching. Therefore, special considerations must be made regarding how to make efficient use of the intermediate
results which are consistent with the current requirements by the user. This is particularly beneficial in exploring the solution space to find another solution by changing the results of the obtained solution as little as possible. We have created an environment for UDS where the desired solution can be obtained interactively from a series of the user’s requirements.

In order to implement such a UDS, we have classified the variables into different groups. We proceed as follows:

Given a solution which is an assignment of values \( x_i \) to all the variables \( \{v_1, v_2, \ldots, v_n\} \) such that a set of constraints are satisfied, we create groups \( \{g_1, g_2, g_3, g_4\} \) consisting of subsets of the variables in the given problem. The four groups are:

- **g1** – Hold group
- **g2** – Base group
- **g3** – Free group
- **g4** – Change group

Thus the variables in \( \{v_1, v_2, \ldots, v_n\} \) can be categorized as follows:

\[
\begin{align*}
 v_{h1}, v_{h2}, \ldots, v_{hn1}, & \quad v_{b1}, v_{b2}, \ldots, v_{bn2}, & \quad v_{f1}, v_{f2}, \ldots, v_{fn3}, & \quad v_{c1}, v_{c2}, \ldots, v_{cn4} \\
 \text{hold group} & \quad \text{base group} & \quad \text{free group} & \quad \text{change group}
\end{align*}
\]

where \( n_1 + n_2 + n_3 + n_4 = n \) and \( 0 \leq n_1, n_2, n_3, n_4 \leq n \). The variables in the hold group should not have their assignments changed. The base group variables should only be changed if it is imperative to find a new solution. In other words, we keep as many as possible of the assignments in the base group unchanged. Even then, a new solution is adjacent to the solution obtained, measured by changing the least number of such group variables. The variables in the free group are free to be changed, ranging over their domains in the normal way. The variables in the change group are free except that they cannot take on their current values in the new solution. In other words, the change group variables must be changed.
**5.5. Variable Classification**

---

**procedure search2(i, v_i)**

; search for a possible solution

begin

1: if i > n then

begin

    output(v_1, v_2, ..., v_n)

    solution_flag = 'yes'

    ; a possible solution is found

end

else if i = 0 then

begin

    solution_flag = 'no'

    print("Sorry, no solution exists")

end

2: for d = each element of D_i do

begin

    v_i ← d

    conflict_flag ← check(i, v_i, C)

    ; invoke constraint.rules to check

    ; whether v_i is consistent assignment,

    ; return 'no' if it doesn't conflict with

    ; constraint sets, otherwise return 'yes'

    if conflict_flag = 'no'

        then search2(i + 1, v_{i+1})

end

---

Figure 5.6: The algorithm of searching for a possible solution for UDS2.
This classification makes UDS more flexible, and hence they provide a very simple interface on which the feasible solution searching can be conveniently monitored and directed in order to converge upon the solution which satisfies the user.

Clearly, CSP techniques can be used to restrict the domain values in the search for a new solution, since the variables in the hold and change groups have new domains. Each variable in the hold group contains only one value in each domain. The number of domain values decreases by one in the change group because the variables in this group cannot take the values in the initial solution. Thus, we eliminate these values from the corresponding domains. CSP techniques can be used to filter the variable domains (Details will be discussed in section 7.5).

The description of the procedure of reorder variable domain in Figure 5.7 line 1 can be found in Chapter 7. Minimal changes in the base group are shown in Figure 5.8.

In Figure 5.7, queue is a data structure which consists of a list of the variable assignments in the base group, with index i standing for the position of variable most recently changed. At the initial call, queue takes a list of the variable assignment in the base group of the solution obtained and 0 which expresses no variable change in the base group. We expand one more variable changed after that position. We remember the position of the variable changed so as to avoid repetitive variable changes.

5.6 Comparison of two strategies

There are two alternative choices to achieve step-by-step UDS of searching for a solution along the user's preferred directions. The former, backjumping, is to back up to the highest level in the search tree. It may inherit some results from previous search, but it cannot always save as many as possible of the obtained results from changes. The later, variable classification, is to make effective use of the results obtained in previous searching. This classification makes user-directed search more flexible and efficient so that the new solution has minimal changes from the solution obtained previously.
5.6. Comparison of two strategies

procedure UDS2

\( g_1 \): set of variables in hold group. \( a_h \) is a current assignment for \( v_h \).

\( g_2 \): set of variables in base group. \( a_b \) is a current assignment for \( v_b \).

\( g_3 \): set of variables in free group. \( a_f \) is a current assignment for \( v_f \).

\( g_4 \): set of variables in change group. \( a_c \) is a current assignment for \( v_c \).

queue: a queue storing the index and the assignments for the variables in \( g_2 \).

\( q \): a queue storing the same information as above which waiting for doing one more variable change.

begin

1: reorder.variable.domain\( (D_1, \ldots, D_n) \); see sub-section 7.4.2

2: input\( (g_1, g_2, g_3, g_4) \); identify the variables in each group

\( g_1 = \{ v_{h_1}, \ldots, v_{h_{n_1}} \} \)

\( g_2 = \{ v_{b_1}, \ldots, v_{b_{n_2}} \} \)

\( g_3 = \{ v_{f_1}, \ldots, v_{f_{n_3}} \} \)

\( g_4 = \{ v_{c_1}, \ldots, v_{c_{n_4}} \} \)

3: \( D'_{h_1} \leftarrow \{ a_{h_1}, \ldots, a_{h_{n_1}} \} \)

4: \( D'_{c_1} \leftarrow D_{c_1} - \{ a_{c_1}, \ldots, a_{c_{n_4}} \} \)

5: \( D_{b_1}, \ldots, D_{b_{n_2}}, D_{f_1}, \ldots, D_{f_{n_3}} \) are reduced to \( D'_{b_{n_2}}, D'_{f_1}, \ldots, D'_{f_{n_3}} \) by calling CSP module

\( \because \) because the variable domains in \( g_1 \) and \( g_4 \) are changed, CSP module is used to reduce the domain values.

6: queue \( \leftarrow ((0, a_{b_1} a_{b_2} \ldots a_{b_{n_2}})) \); contains the index and the assignments for the variables in the base group

7: \( q \leftarrow \text{nil} \)

8: while (not (null queue))

\( \begin{align*}
8.1: \ & w \leftarrow \text{pop}(\text{queue}); \\
8.2: \ & q \leftarrow \text{push}(w) \\
8.3: \ & \text{order}(v_{h_1}, \ldots, v_{h_{n_1}}, v_{b_1}, \ldots, v_{b_{n_2}}, v_{f_1}, \ldots, v_{f_{n_3}}, v_{c_1}, \ldots, v_{c_{n_4}}) \\
8.4: \ & \text{search2} \ (f_1, v_{f_1}) \\
8.5: \ & \text{if solution_flag = 'yes'} \ \\
\ & \quad \text{then return('yes')} \quad \text{another solution is obtained} \\
\ & \quad \text{else if (null queue) then queue \leftarrow one.more.change} \ (q) \\
\end{align*} \)

9: return('no'); no solution

end

Figure 5.7: Algorithm for UDS2.
one.more.change (qu) ;return a queue
   \( a_{b_1}, a_{b_2}, \ldots, a_{b_{n_2}} \) are values for \( v_{b_1}, v_{b_2}, \ldots, v_{b_{n_2}} \) respectively.

begin
1: if (null qu) then return(nil)
2: w ← pop(qu)
3: q1 ← nil
4: i ← first element in w ;i is index for the variable in
   ;the base group most recently changed
5: n2 ← the number of the variables in the base group
6: for j= i+1 to n2
   for d = each element in \( D_{b_j} - \{ a_{b_j} \} \) do
      conflict_flag ← check \((b_j, v_{b_j}, C)\);invoke constraint.rules
      if conflict_flag = 'no'
         then q1 ← push(j \((a_{b_1}, \ldots, d, \ldots, a_{b_{n_2}})\))
            ;where d replaces \( a_{b_j} \)
7: return(q1);
end

Figure 5.8: The algorithm of closeness change on the variables in the base group.

In the following chapters, we will show implementation of the CSP techniques
and the two above-mentioned strategies in UDS to practical problems to improve
search efficiency and effectiveness.
Chapter 6

Case Study I: Truck Dispatching Problem

6.1 Introduction

Truck dispatching problem [Filman 88] is highly constrained by the finite set of resources which are to be shared and efficiently used, and the various complex relationships which are to be satisfied. Usually, the development of a truck dispatching problem solver, for users to assign tasks to drivers and trucks, can be described as a problem where a feasible solution must satisfy certain requirements and relations in terms of a set of constraints. These requirements and relations range from those common sense, for example,

"You can’t put more on a truck than it can hold."

to the legalities of this specific problem, for example,

"A driver with a licence lower than class 3 must not drive a truck that requires a driver with at least class 3 licence".

As an example, we consider a case of UDS where a set of solutions can be found, and only one is required as long as the user thinks it acceptable. If a solution is found, and it is returned to the user through interface, the user can decide whether it is satisfactory or not. If not, the user may wish to change some constraints, facts, or information according to the returned solution and then start the search for a new
solution of the present requirements. This process is continued until an acceptable solution is obtained. Our main concern is to achieve the highest possible efficiency in a search which converges on a user-desired solution. One simple approach we have used is to translate the user's present requirements into appropriate additional 'constraint rules'. Steps are taken to ensure that the next solution can be sought in a way which avoids the exploration of the decisions which already exist in the search tree. The shortcoming of this approach is that it considers all the possibilities for the most recent choice before backtracking any earlier decision. In order to remedy this, we utilize ATMS and KEEworlds systems in KEE. When a new requirement is given and the search is restarted, some steps are taken to begin further searching by backing up to the highest level in the search tree. Thus, in searching for the next solution, greater efficiency may be obtained by avoiding unnecessary backtracking and by using the information and results obtained in the previous solution.

Furthermore, in order to improve search efficiency of our UDS truck dispatching system, we perform data pre-processing – using CSP techniques (described in Chapter 3 and developed in Chapter 4) – to delete domain values which will not occur in any part of the possible solutions. These values violate certain constraints and thus will lead to dead-end searching. By this data pre-processing step, we can avoid this useless searching and thus explore a smaller search space. For instance, in our problems, we can eliminate some candidate trucks and drivers which violate a corresponding set of constraints prior to solution searching. On one hand, the search space is reduced and, on the other hand, the dead-end backtrackings related to those deleted candidates are avoided. We have found that the search efficiency in our system can be significantly improved by performing the data pre-processing.

Another technique we have used to improve search efficiency is backtracking information processing – constraint recording. In this step, some particular information about dead-end searching will be recorded in the form of an additional ‘constraint’. By implanting the ‘constraint’ in the knowledge base, we can avoid repeating the same type of dead-end searching in further solution search. Thus, the search space will be smaller than that required by the standard backtracking search.
The backjumping approach cannot always guarantee the transfer of as much information as possible from the previous solution to the one expected by the user. In the next chapter, we will present a special approach to deal with this in the case of a crossword puzzle problem.

6.2 Problem Description

To start our discussion, we give a description of a truck dispatching problem which we will solve using the above-mentioned methods. Given the required trips, the available trucks and drivers, a dispatcher has to assign to each trip a suitable driver and a truck. That is, the dispatcher devises a schedule — a collection of trips (an assignment of a truck and a driver to a particular itinerary) such that all shipments are picked at their origins and delivered to their destinations.

The activity of the dispatcher is to create a schedule. A schedule is a set of trips, where each trip specifies: (1) a truck, (2) a driver, and (3) an itinerary: an origin for that trip and a series of actions, where each action is a triple: a place to go, a shipment to operate on, and an action to take (put shipment on truck, take it off, just visiting). For example, (Melbourne computers on) means that computers were put on truck in Melbourne.

Each trip, truck and driver arranged in the schedule must obey a set of constraints — static and dynamic constraints. Each constraint is a restriction on a trip, a driver, a truck or a collection of them that specifies a legal state. We take a heuristic search approach to generate the schedule. One feature of our approach is that we can represent and use a variety of different types of constraints to control and guide the search.

6.2.1 Static Constraints

The first step in the construction of the truck scheduling problem solver is to determine the categories of constraints a dispatcher considers and to represent them. The static constraints are general restrictions satisfied by all the possible solutions. That
is, each possible solution must satisfy all these static constraints. The constraints come from:

- The common sense. For instance, "You cannot put goods on a truck which exceed the maximum weight and volume capacity of the truck." and "The location of the driver of a trip must be the same as the location of the truck of a trip." and so on;

- The legalities of this given problem. Such as, "The driver needs the right kind of licence to drive a particular truck."

These kinds of constraints can be represented in the knowledge base as deduction rules. A static constraint set is set up when creating knowledge base (KB) for storing general data and rules.

### 6.2.2 Dynamic Constraints

Dynamic constraints come from the new requirements from the user, such as some conditions which have not been considered beforehand, through the UDS. This kind of constraint occurs when the user is not satisfied with the feasible solution found or when some circumstances happen. For instance, the dispatcher may prefer driver White to drive a particular truck. But the system has already assigned him to drive another truck. So the solution needs to be modified to suit the user's requirements. In another case, White is assigned to drive the truck T1 for Trip.1. However, when the circumstance happens, such as, White is ill, the system can not assign him to any trip. In this case, a new solution needs to be found to satisfy this exceptional requirement. How do we find a new solution efficiently and effectively? In one simple approach, all the possible solutions are found using the initial set of constraints and are recorded. They are used when new requirements arrive. Practical considerations reveal that finding and recording all the possible solutions may be too costly and may involve unnecessary work. In such cases, it may be worthwhile to find only one possible solution and then to utilize the information obtained in the previous solution when a new requirement from the user arrives. That is to make maximum
6.3. System Overview

The truck dispatching problem is implemented here by using KEE. This application is to illustrate the integration of KEE, CSP module and other methods for interactive problem solving. In particular, it applies to CSP module (developed in Chapter 4) in KEE to perform pre-processing. It also integrates rules and Lisp methods to realise searching for a possible solution. When a solution is found, it will be sent to the user's interface for the user to decide whether the solution is satisfactory. If the solution is considered unacceptable, the user may look forward to obtaining another possible solution without change of requirements or with additional requirements. We have built menus to ask the user for the additional conditions if the user intends to present them. The requirements will be transformed into deduction rules which can allow for the creation of ATMS justifications. We can take advantage of using ATMS justification and worlds to backtrack from the highest level in the search tree to carry out a further search for a new solution. Our prototype of the truck dispatching problem also uses active values to modify certain relevant knowledge automatically, and provides menus to inquire of the user for additional requirements.

The basic searching for a solution to the truck dispatching problem is carried out via forward chaining.

6.3.1 Flow of Control

Figure 6.1 shows the flow of controls of our truck dispatching system. In initialization, the information about trips, trucks and drivers is put into the knowledge base. A user can select candidate trucks and drivers or the system will automatically take all the trucks and drivers in the knowledge base as candidate trucks.
and drivers by invoking proper rules. A KEEworlds window is created to show the searching process and intermediate results during searching. Our system will carry out data pre-processing using CSP module (developed in Chapter 4) to reduce the search space in advance of solution searching. Then dispatcher.assistant.rules and constraints.rules will be invoked to search for a possible solution. When a solution is found, it is sent to a window interface designed for solution display. If the results are considered unacceptable, the user can add additional constraints to the problem under consideration. The added constraints will be transformed into deduction rules which can be used to find the highest level in the search tree to continue searching for a new solution.

6.3.2 Design Features

A number of useful design features have been incorporated into our system. Among them the important features are:

- Data pre-processing: A dispatching problem involves generating a feasible solution satisfying an irregular set of constraints. It can be expressed as a constraint satisfaction problem. Some of its constraints can be formulated as unary and binary predicates in CSP. Thus we can employ CSP techniques to perform data processing prior to searching for solution. This process is called data pre-processing. It uses the CSP module (see Chapter 4) to drastically reduce the amount of searching that is required to solve a truck dispatching problem without data pre-processing.

- The intertwining of rules and Lisp in reasoning: Our system embeds rules, which are used for its basic searching, in a nest of Lisp, which is used for the procedures that establish the flow of control and to send messages to create menus to ask the user for information needed by the system during searching. It is important to effectively use rules and methods (Lisp procedure) together to search for a possible solution efficiently.
6.3. System Overview

Figure 6.1: The control structure of a truck dispatching system.
Also we group the rules into rule classes (for details see sub-section 6.6.2). These limit the number of rules to be considered in rule-based reasoning. This allows the system to avoid use of excessive CPU time in the usual pattern-matching routines.

- The use of menus to ask the user for information needed by the system during reasoning;

- Constraint recording: storing the reasons for the dead-end so that the same conflicts will not arise again in the continuation of the search.

- Automatic change of relevant knowledge when certain information is modified via active values.

- The utilization of ATMS, KEEworlds, and the rule system allows choices to be changed independently of the order in which were made. In other words, the search can jump back to the highest point in the search tree to continue searching for a new solution when new requirements are provided by the user.

### 6.4 Knowledge Representation

The first step in constructing an artificial intelligence program is to build a knowledge base. In order to act intelligently, a system must have knowledge about the domain of interest. Therefore, the key to the success of any AI system is a proper choice of the knowledge representation that best fits the domain knowledge and the problem to be solved. The choice of a particular knowledge representation scheme for a given problem strongly affects the efficiency of constructing a solution for the problem under consideration.

#### 6.4.1 Truck Dispatching Knowledge

Truck dispatching knowledge representation must be capable of representing domain-specific knowledge, such that the "intelligent" system can make good use of the knowledge in performing its intended task efficiently and effectively.
6.4. Knowledge Representation

Figure 6.2: Trucks hierarchy for representing all the trucks.

We represent the objects of truck scheduling as units and arrange these units in a class hierarchy. We create the following classes of objects in our domain: Drivers, Trucks, Trips, Licence.classes and so forth. Trucks have the subclasses of Big.Trucks, Medium.Trucks, and Small.Trucks. Individual trucks are members of these classes (Figure 6.2). In Figure 6.2, a solid line represents a class-subclass relationship, like that between the units Trucks and Small.Trucks. Trucks represents the most general classification in its hierarchy. Small.Trucks is a subclass of Trucks. That is, it is less general. The dashed line between the units represents a class-member relationship, such as that between the units Small.Trucks and a particular truck, T4. The properties in Trucks may then be inherited by its subclasses such as Small.Trucks and its members like T4.

Describing the attributes of trucks, such as the location of each truck, in the knowledge base is accomplished with slots within a unit. Slots can store numerical and textual data about the object as well as more complex information, such as procedural programs that can execute Lisp code or begin rule chaining.

We define the following relations on objects in these objects. The numbered pairs marked <> in Table 6.1 represent the minimum and maximum cardinalities of the relation. Cardinality is referred as the number of values allowed in a slot. Using cardinality restrictions can provide protection against having illegal values installed in a slot. For example, in the Trucks unit, the slot named location represents the
Chapter 6: Case Study I: Truck Dispatching Problem

<table>
<thead>
<tr>
<th>Relation</th>
<th>Cardinality</th>
<th>default</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>license.class(trucks,</td>
<td>&lt; 1, 1</td>
<td>class.3</td>
<td>The class of license required for a driver to drive this truck.</td>
</tr>
<tr>
<td>license.classes)</td>
<td></td>
<td>class.2</td>
<td></td>
</tr>
<tr>
<td>big.trucks</td>
<td></td>
<td>class.1</td>
<td></td>
</tr>
<tr>
<td>medium.trucks</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>small.trucks</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>location (trucks, cities)</td>
<td>&lt; 1, 1</td>
<td></td>
<td>The location of the given truck.</td>
</tr>
<tr>
<td>volume.capacity(trucks,number)</td>
<td>&lt; 1, 1</td>
<td>1200</td>
<td>The volume capacity of a truck.</td>
</tr>
<tr>
<td>big.trucks</td>
<td></td>
<td>640</td>
<td></td>
</tr>
<tr>
<td>medium.trucks</td>
<td></td>
<td>400</td>
<td></td>
</tr>
<tr>
<td>small.trucks</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>weight.capacity(trucks,number)</td>
<td>&lt; 1, 1</td>
<td>32000</td>
<td>The weight capacity of a truck.</td>
</tr>
<tr>
<td>big.trucks</td>
<td></td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td>medium.trucks</td>
<td></td>
<td>5000</td>
<td></td>
</tr>
<tr>
<td>small.trucks</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: The attributes of Trucks.

location of a given truck. The slot should contain at most one value, because a given truck is located in one place. By setting the maximum cardinality to 1, we can avoid having more than one location accidentally entered in the location slot. However, since the model is explicitly a partial model, it is possible for it to have less than the minimum cardinality values because the system assumes that more values may be put into the slot later: the unit may not yet be completed, or the values in the slot may be changed.

Since the task of the dispatcher is to assign an available truck and a driver to a particular trip, the knowledge base needs to keep relevant information about trucks, drivers and trips which are required for solving the problem. Table 6.1 shows the information about a truck which contains at least licence.class, location, volume.capacity and weight.capacity. The practical data about a particular truck is given at the initial stage.

Table 6.2 shows the information about a particular truck T4 in the knowledge base. T4 is a member of the class Small.trucks. So it inherits some slots and default values from Small.trucks and Trucks.

All the drivers are members of the class Drivers. There are three types of li-
6.4. Knowledge Representation

The T4 Unit in TDES Knowledge Base

<table>
<thead>
<tr>
<th>Own slot: licence.class from T4</th>
<th>Inheritance: OVERRIDE.VALUES</th>
<th>Valueclass: LICENSE.CLASSES</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Comment: &quot;The class of licence required for a driver to drive this truck.&quot;</td>
<td>Values: class.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Own slot: location from T4</th>
<th>Inheritance: OVERRIDE.VALUES</th>
<th>Valueclass: CITIES</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Comment: &quot;The location of this truck.&quot;</td>
<td>Values: Woden</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Own slot: volume.capacity from T4</th>
<th>Inheritance: OVERRIDE.VALUES</th>
<th>Valueclass: NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Comment: &quot;The volume capacity of a truck.&quot;</td>
<td>Values: 400</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Own slot: weight.capacity from T4</th>
<th>Inheritance: OVERRIDE.VALUES</th>
<th>Valueclass: NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Comment: &quot;The weight capacity of a truck.&quot;</td>
<td>Values: 5000</td>
</tr>
</tbody>
</table>

Table 6.2: Details of the truck T4.

<table>
<thead>
<tr>
<th>Relation</th>
<th>Cardinality</th>
<th>default</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>licence.class(drivers, licence.classes)</td>
<td>&lt; 1, 1 &gt;</td>
<td>default</td>
<td>The class of licence a driver holds.</td>
</tr>
<tr>
<td>location (drivers, cities)</td>
<td>&lt; 1, 1 &gt;</td>
<td>default</td>
<td>The location of this driver.</td>
</tr>
<tr>
<td>maximum.driving.time (drivers,number)</td>
<td>&lt; 1, 1 &gt;</td>
<td>default</td>
<td>The maximum driving time for a driver in a day.</td>
</tr>
</tbody>
</table>

Table 6.3: The attributes of Drivers.

ence classes, namely, class.1 class.2 and class.3, which are members of the class Licence.classes.

Table 6.3 shows the information about Drivers. For each driver, it takes a number of slots to keep the associated attributes, such as, licence.classes, location, etc.

As shown in Table 6.4, some attributes about the trips such as itinerary, origin, max.volume, and max.weight, are put into the knowledge base at the initial stage. The rest, such as, driver and truck are assigned to a suitable truck and driver during the problem-solving process.

We need also to set up rules for our problem solving. In the next section, we
Chapter 6: Case Study I: Truck Dispatching Problem

<table>
<thead>
<tr>
<th>Relation</th>
<th>Cardinality</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>driver(trips,drivers)</td>
<td>&lt; 1, 1 &gt;</td>
<td>The driver for this trip.</td>
</tr>
<tr>
<td>itinerary(trips, itineraries)</td>
<td>&lt; 1, 1 &gt;</td>
<td>The itinerary of this trip.</td>
</tr>
<tr>
<td>truck(trips, trucks)</td>
<td>&lt; 1, 1 &gt;</td>
<td>The truck for this trip.</td>
</tr>
<tr>
<td>origin(trips, cities)</td>
<td>&lt; 1, 1 &gt;</td>
<td>The origin of the itinerary of the trip.</td>
</tr>
<tr>
<td>max.volume(trips, number)</td>
<td>&lt; 1, 1 &gt;</td>
<td>The largest volume of goods on this trip.</td>
</tr>
<tr>
<td>max.weight(trips, number)</td>
<td>&lt; 1, 1 &gt;</td>
<td>The largest weight of goods on this trip.</td>
</tr>
</tbody>
</table>

Table 6.4: The Relations of Trips.

give a detailed description of rules for the dispatching problem.

6.4.2 Constraint Rules and Dispatching Rules

1. constraint rules:

A constraint rule is a constraint from the practical problem – it is presented in terms of rules in our case. A constraint rule may affect a schedule. It may determine the admissibility of a schedule, or it may determine the acceptability of a schedule. Thus, in the construction of the solution to the truck dispatching problem, it is necessary to establish the constraint rules from the constraints which a dispatcher considers as basic and general restrictions to be obeyed. These constraints include:

(1). Constraints without exceptions (static constraints) – Each truck has physical constraints, such as location, volume capacity, weight capacity and so forth, prohibiting it from serving on certain trips, for example, "you cannot put more on a truck than it can hold". Obviously, such constraints must be satisfied under any circumstances. They are defined in the initial environment. Implicitly, if we mention two trips, t1 and t2, we assume that they are part of the same schedule. We represent the constraints in positive form; that is, the situation violates the constraints if the given WFF is not true. Some of such constraints are as follows:

- A driver can only take a single trip.
6.4. Knowledge Representation

\[ \forall t_1, t_2. \quad driver(t_1) = driver(t_2) \Rightarrow t_1 = t_2 \]
where \(driver(t_i)\) means the driver for \(t_i\).

- Similarly for trucks – each truck can only go on a single trip.
\[ \forall t_1, t_2. \quad truck(t_1) = truck(t_2) \Rightarrow t_1 = t_2 \]
where \(truck(t_i)\) means the truck for \(t_i\).

- One can’t put more (volume or weight) on the truck than it can hold.
\[ \forall t. \quad max.volume(t) \leq volume.capacity(truck(t)) \]
\[ \forall t. \quad max.weight(t) \leq weight.capacity(truck(t)) \]
where \(t\) stands for any trip in the schedule.

- The licence rule – the licence class of a truck should be less or equal than that of a driver driving it.
\[ \forall t. \quad licence.class(driver(t)) = m \land licence.class(truck(t)) = n \Rightarrow m \geq n \]

We express all these constraints as constraint rules in KEE. For instance, ‘one cannot put more volume on the truck than it can hold’ can be expressed as follows in KEE:

\[
(\text{while} \\\n(\text{?t is in class trips}) \\\n(\text{the max.weight of ?t is ?m}) \\\n(\text{the volume.capacity of (the truck of ?t) is ?w}) \\\n(\text{lisp (\text{> } ?m ?w})) \\\n\text{believe} \\\n\text{false } ))
\]

(2) Constraints according to user’s requirement (dynamic constraints) – Users can add more constraints during the course of searching a new feasible solution. When a possible solution is obtained and it is unsatisfactory, the user may add more particular constraints, such as White must be assigned to drive T2. The
new solution to be sought must satisfy a larger set of constraints including both the static and the dynamic constraints.

Generally, all the possible solutions must satisfy all static constraints as well as some dynamic constraints based upon the user's requirements.

2. dispatching rules:

To make a decision, the system uses two types of production rules: action rules and deduction rules. The first represents dispatching rules. The second represents constraint rules. Dispatching rules determine that, when there are candidate trucks and drivers, it is appropriate to assign the truck and driver for a particular trip and permit the system to search the next level of rules (constraint rules) to obtain an acceptable assignment. They also rule out some trucks or drivers that are unacceptable choices for a given trip.

The rule for dispatching a truck for a trip is represented in KEE as follows:

(assign.truck
  (if (the trip.list of assist is ?l)
      (not (equal ?l nil))
      (equal ?first (car ?l))
      (a candidate.truck of assist is ?v)
      (cant.find (the nogood.truck of ?first is ?v))
  then in.new.world
      (delete (a candidate.truck of assist is ?v))
      (change.to (the truck of ?first is ?v)
        using
        constraints.rules)
      (a pending.trip of assist is ?first))

In plain English, this rule is - "If the trip list is not empty, and there is at least one candidate truck, then (1) remove that truck from the set of candidate
6.5. CSP for Data Pre-Processing

Our truck dispatching problem may be considered as a kind of constraint-satisfaction problem (CSP). These are usually difficult because they have to satisfy an irregular set of constraints while working in very large search spaces. It is important to perform data processing prior to the solution search. Thus the amount of searching required can be greatly reduced. We accomplish this by using the CSP module developed in Chapter 4. Pre-processing can shield the searching system from the mass of fruitless data collected in the initial environment. It can be used to filter the domain values according to static and dynamic constraints. Thus it is important to perform pre-processing before searching.

Constraints in CSP may be specified extensionally by enumerating all consistent values, or specified implicitly as functions. In our dispatching system, the trips, drivers, and trucks are represented by variables where each has a domain of possible values and corresponding constraints. We specify constraints implicitly as functions. We use three variables: trip, driver and truck, specifying candidate trips, drivers and trucks respectively.

We will discuss the following:

1. What kinds of domains allow us to apply such pre-processing in our case?
2. How to do the pre-processing?
3. With an example, what we can achieve by performing data pre-process?

We are concerned now with a class of pre-processing algorithms which reduce the problem domains and transform a given constraint network into a more explicit representation before it is subjected to a search algorithm for a solution. Data pre-processing performed by CSP techniques amounts to reducing the search space, with the aim of eliminating some candidate domain values which cannot become part of
any possible solutions. As a result, much of the processing which is unnecessary during the search can be avoided.

We now explain how to effectively and efficiently incorporate pre-processing to solve the truck scheduling problem. In this problem, there are various kinds of constraints. Some constraints can be used to eliminate some candidate trucks and drivers which cannot satisfy initial constraints.

An example constraint network for our problem is shown in Figure 6.3. This is a binary constraint network whose nodes are variables trip, driver, and truck and whose variable values are the possible trips, drivers, and trucks respectively. A link between two variables represents the set of value-pairs permitted by the constraint between the variables.

Before any attempt is made to search for a possible solution, we apply CSP techniques to eliminate some values of Trucks and Drivers to yield a smaller search space. For instance, for a driver, if there does not exist any truck to satisfy the constraints between it and him, he can be deleted from the driver domain immediately. The search will never try this driver. Also, we can delete any candidate driver from driver domain, if this driver does not satisfy the following licence constraint:

\[(\forall d)(d \in Driver)(\exists truck)(truck \in Truck)\]

\[\text{licence.class}(d) \geq \text{licence.class}(truck)\]

The above operation can be implemented by applying consistency checking between Driver and Truck in the CSP module (in Chapter 4). We can perform the same consistency check between Truck and Trip, and between Driver and Truck.
we can eliminate as many as possible of the values of variable domains in advance of the search. Therefore, a number of unnecessary searches are avoided.

Now we give a simple example to show some strong points of data pre-processing. Assume the driver domain is \{Brown, Gray, Green, White\} and the truck domain is \{T1, T3, T2, T4\}. The itineraries required for drivers and trucks are (Trip.1, Trip.3). Figure 6.4(a) shows the search space for finding a possible solution without performing pre-processing. Figure 6.4(b) shows the initial data regarding candidate trucks and drivers, while Figure 6.4(c) shows the candidate trucks and drivers after data pre-processing, and Figure 6.4(d) shows the search space after performing pre-processing.

### 6.6 Constraint-Based Search

After performing pre-processing to reduce the search space, a constraint-based search is started to search for a possible solution. When a solution is obtained and returned through an interface, the user can determine whether the search was satisfactory or
not. If it was, the current solution is accepted as the final solution. If it was not, the current solution is rejected and a new solution must be found. The user can show his preference by intervening and directing the process of searching through UDS.

6.6.1 Constraints and Search

The search begins at the specified initial state where the value of trip.list slot in assist unit contains a list of all the trips needed to be assigned a suitable truck and driver. Invoking the specified action rules extends this initial state. After each application of an action rule, the generated states are checked by applying the constraints, and only those states which do not conflict with the constraints are kept for the next iteration of rule application. A complete dispatching scheme is defined by the path from initial state to the end state in the search space.

As the search proceeds, it continually tests whether the goal state has been generated, and whether the search has come to a dead-end without finding a feasible solution. The goal state, which gives a possible solution, has been reached when the value of trip.list slot in assist unit is nil. That means, all the trips have been assigned to suitable trucks and drivers. However, if the solution is unsatisfactory, then the system should continue searching for a new solution.

In general, problem solving is a process of finding, or constructing, a solution to a problem based on certain set of beliefs. These beliefs can be in the form of the values of some variables, a complex data structure, or a collection of formulae. Note that different set of beliefs is believed at different points (contexts) in the problem-solving process.

KEEworlds is an excellent facility for managing problem-solving involving a search. Whenever a problem requires the examination of a decision space of considerable complexity that might require backtracking, KEEworlds is an appropriate and useful tool. For such problems, KEEworlds permits the generation and storage of intermediate stages in the search space. If the exploration of a given branch of the search tree proves unfruitful, the KEEworlds allows the search to continue without returning to the top of the search space. Saving the intermediate stages of the search
space allows backtracking to the intermediate result and continuation of the search from that point. We use the appropriate rules and the Assumption-Based Truth Maintenance System (ATMS) to realize backjumping in our system.

The ATMS in KEE sets up dependencies between facts in the knowledge base. We use ATMS to represent constraint rules and to detect contradictions. That is, we express constraints as deduction rules whose premises are the negative of the constraints and whose conclusions are false. In our truck dispatching problem, there are a number of constraint rules, such as weight capacity restrictions. This constraint, which we know quite well, may be represented as:

\[
(\text{Truck.weight.limits}
\begin{aligned}
\text{(while)} & \\
 & (\text{the max.weight of } ?t \text{ is } ?m) \\
 & (\text{the weight.capacity of (the truck of } ?t) \text{ is } ?w) \\
 & \text{(lisp } (> \ ?m \ ?w)) \\
\end{aligned}
\text{believe}
\text{false })
\]

where \( ?t, ?m \) and \( ?w \) are variables which can be instantiated by a trip, the maximum weight on the trip and weight capacity of a truck, respectively. Such a constraint will be represented by a justification in the ATMS. When the premises are true, its conclusion is put into the system. We can use the justifications of contradictory conclusions to determine which assignment conflicts with the constraints. One of the purposes of establishing and maintaining these justifications is to allow, if a contradiction is detected, to backtrack along the searching, to change the assumptions in some way, and to search forward. More constraint rules relating to the Truck Dispatching Problem are given in appendix A.

6.6.2 Rule-Based Reasoning

Generally speaking, rules traditionally are used to represent heuristics or “rules of thumb”. They are the only way of representing unordered declarative information
in a knowledge-based system. Such information is used for either stating new facts to the system and letting it determine the consequences of these facts (forward chaining) or by asking the system about a particular fact and letting it determine whether or not that fact can be deduced given the facts that the system already knows (backward chaining).

Searching for a possible solution in our system is based on forward chaining. KEE supports the use of two different types of rules, deduction rules and action rules. Deduction rules create justifications for facts which are used to represent constraints in our application. Action rules, on the other hand, cause some actions (e.g., adding facts, deleting facts, executing a procedure and so on) to be carried out.

There are two kinds of action rules, namely, *same world action rules* and *new world action rules*. As the name suggests, same world action rules run in a single world. For instance, in our problem, *Make.candidate.drivers* and *Make.candidate.trucks* in Figure 6.6 are same world action rules.

New world action rules are useful to model *optional* actions, sequences of states, and alternative hypotheses. If the premises of a new world action rule are true, a new world is created into which the conclusions of the rule are asserted. Usually, a new world action rule makes all the changes in the new world. *Assign.driver* and *Assign.truck* are examples of such new world action rules (see Figure 6.5).

Understanding the behaviour of the search for a solution to a truck dispatching problem requires understanding the scheduling algorithm of the rule system in KEE. A rule system cycles through a three-step process of

1. determining which instantiations of rules are eligible to fire;
2. selecting a particular instantiated rule to fire; and
3. taking the action required by that firing.

The system contains action rules in addition to the constraint rules. The unit *assist* keeps its local search state on the *candidate.trucks*, *candidate.drivers*, *trip.list*, *pending.trip* and so on. At any point in the search, the *candidate.trucks* and *candidate.drivers* slots contain the available but not-yet-assigned trucks and drivers, the *trip.list* slot keeps a list of the trips that have not yet been assigned a suitable truck
Figure 6.5: The rule hierarchy used for our truck dispatching problem.
and a driver. We put a trip which has already been assigned a truck but has not yet been assigned a driver in the pending.trip slot of the assist unit. We mark as satisfied in satisfy slot of assist unit when all the trips have trucks and drivers satisfying all the constraints.

The search begins by creating a world named start with an initial state which triggers the consideration of the Make.truck.driver.rules. The member of this class, namely Make.candidate.drivers and Make.candidate.trucks, will be fired. These two rules can be applied since their premises have been satisfied. The purpose of these two rules is to get candidate trucks and drivers. Then the new world action rules, Assign.truck and Assign.driver will come into effect. New world action rules, when applied, create a new world in which to record their effects. The primitive action operators are implemented by using new world action rules in the forward direction. Assign.truck selects the first trip in the trip.list slot of the current world, finds a candidate truck if a candidate truck is available, and then executes the following actions in the new world:

(i). assigns that truck to the trip. The constraint rules then get their turn. We invoke constraint rules on different rule classes from within forward chaining. If the assignment conflicts with constraints, we make this world nogood and backtrack to an earlier stage. Otherwise, go to (ii);

(ii). marks that trip in pending.trip waiting for a driver to be assigned;

(iii). removes that truck from the set of candidate trucks.

The above illustrates the function of the Assign.truck rule. Then Assign.driver is fired to assign a driver for the trip in pending.trip, which has been marked.

When all legal truck and driver assignments are finished, the value in trip.list slot will become empty and the condition of Ask.user rule becomes true. The conclusions of Ask.user cause messages to be sent (carry out procedure) and cause the creation of menus to inquire of the user for his decision on whether the solution can be accepted or not.

The Stop.all rule stops the system after finding a feasible solution satisfying all the constraints and user’s requirements. In Figure 6.6, each world corresponds to a
6.7 Constraint Recording While Searching

The subject of improving search efficiency has been on the agenda of researchers in the domain of AI problems for quite some time [Montanari 74], [Mackworth 77], [Mackworth 84], [Haralick 80], [Dechter 85]. Some strategies such as those called "intelligent backtracking", "selective backtracking", and "dependency-directed backtracking" have been presented with the purpose of improving search efficiency.

We concentrate on our strategy — constraint recording, namely, analyzing and storing the reasons for a dead-end searching, and then using them to guide the future decisions, in order to avoid repeating the same useless search. That is, we remember what has been done so that the same mistakes will not arise again in the future search. The task of constraint recording is to remember some information in some way during the search and use it in the later search. We show how to use constraint recording to improve search efficiency in our truck dispatching problem. We have investigated the trade-offs between the amount of constraint recording and the improvement in search efficiency. Constraint-recording has the following properties in our case.

1. It works by observing the performance of processing on any given input and recording some relevant information during the search.

2. The overall performance of the system is improved when it is used in conjunction with constraint recording.

3. When the search terminates, the information accumulated by the constraint recording is a part of a new, more knowledgeable, representation of the same problem. That is, even if the processing is invoked once again on the same input, it will have a better searching efficiency.

In our truck dispatching application, we create two slots trip.nogood.driver and trip.nogood.truck in each trip unit. These are used to keep the reason of dead-end
Figure 6.6: An example of the information accumulated during searching.

searching. For instance, the reason for a dead-end searching is that a particular truck T5 cannot be assigned to Trip.1 because of volume limit. Then when reaching that dead-end, the truck T5 is stored in the trip.nogood.truck slot in Trip.1 unit.

We also create a truck.nogood.driver slot in each truck unit to keep information on drivers who cannot serve the particular truck. Also, we set up driver.nogood.truck slot in each driver unit for storing the trucks which cannot satisfy the constraint between a driver and a truck.

In Figure 6.6, a black square inside a node denotes a dead-end in the search process. Figure 6.6 (b) shows the knowledge base remembering T4 in trip.nogood.truck slot of Trip.1 unit when it first meets the dead-end. The reason for this dead end is that truck T4 was assigned to Trip.1, conflicting with the corresponding constraint. Figure 6.6 (c) and (d), respectively, show the knowledge base recording another two dead-ends in the search space. Our system can prevent the truck T4 from being
6.8. Backjumping for UDS

assigned to Trip.1 in the continuation of searching. Thus, useless search, because of assigning T4 to Trip.1, no longer happens in the future search.

Moreover, the knowledge accumulated by recording can be considered as a new part of the knowledge of the system. Thus, when the process of searching for a possible solution is executed once again with the same input, it can achieve a better search efficiency. As Figure 6.6 (e) shows, any dead-end has its reason recorded so that the same conflicts will not arise again.

It is worth noting that when the information of a driver, such as the location of the driver was changed, the stored information relevant to this driver, such as driver.nogood.truck, must be changed too because it may not conflict with certain constraints. Thus, we utilise active values which attach to the corresponding slots. Active values allow us to specify that a particular action should occur every time when an attached slot’s value is modified. For our problem, we attach driver.av to the location slot in drivers unit. Thus, when the location of a driver is revised, the corresponding stored information is modified automatically. The same is true for other data, such as location slot in trucks unit and the origin slot in trips unit, which are attached with active values.

6.8 Backjumping for UDS

Assume we find a possible solution S1 on the base of a static constraint set. If the user is not satisfied with the solution or certain requirements are added, another solution has to be searched using a larger constraints set (initial constraints as well as the new requirement). For instance, suppose it is required by the user that the truck T3 is assigned for the trip Trip.4. A new solution has to be searched upon the user’s request. One method is finding and recording all the solutions to the problem. This method is very costly because of much unnecessary work and, in particular, it cannot be efficiently used when new requests arrive. The other method for solving such a problem is normal backtracking. That is, consider all the possibilities for the recent choice before revising any earlier decision. The disadvantage of this method is that the decision to be revised may not be a choice that contributed to the failure.
Our strategy is to find the highest level in search tree which contributed to the conflict with the user's new requirements and to start further searching from there. By using this strategy the amount of useless backtracking can be reduced. We call this process backjumping. We do the following: when new requirements arrive, we transform them into deduction rules which can create ATMS justifications. For instance, if user requires that the truck T3 must be used for a particular trip Ttip.4, then we transform this constraint into the following deduction rule:

\[
\text{while} \\
\quad (\text{the truck of Trip.4 is } ?t) \\
\quad (\text{not (equal } ?t \text{ T3)}) \\
\text{believe} \\
\quad \text{false}
\]

We set up this kind of rule as a member of the class user.constraint.rules. We specify these rules with the highest privilege. That is, we invoke them before any other rules. Deduction rules such as the above will be invoked and, accordingly, TMS justifications will be created. The ATMS justifications can be used to find which world conflicts with the user's requirements in the previous solution. As we described in Chapter 2, our system uses worlds to represent the decision points in the search tree. If the facts of a world satisfy the conditions of deduction rules, the conclusion false is added to the world, which then becomes inconsistent. The system will avoid those worlds which are inconsistent from further consideration. Thus, we can invoke User.constraint.rules which consists of deduction rules to find the right point for further search based on the user's request. By this process, we can avoid a number of unnecessary backtracks. That is, the system can now continue searching for a new solution from the highest level in the search tree.

An example of finding a new solution satisfying the user's requirements is shown in Figure 6.7, where (a) shows the search process for initial solution S1 satisfying static constraints, and (b) is the output results obtained in solution S1. The user does not accept the solution S1, and it is required that the truck T4 cannot be used for some reason. We create a deduction rule to represent the constraint upon the
6.8. Backjumping for UDS

(a)

(b)

(c)

(d)

Figure 6.7: An example of backjumping in UDS.
user's request as follows:

(User.constraint.rule.2
(while
    (the truck of ?t is T4)
    believe false ))

Figure 6.7 (c) illustrates that the highest level is the decision point TRUCK-TRIP.2-T4. The search is directly back up to the right place to continue the search for a new solution. Figure 6.7 (c) shows the searching process for a new solution and (d) reports the result.

6.9 Summary

Based on the considerations and techniques developed in the earlier chapters, here we have created and implemented a prototype truck dispatching problem solver in KEE. A step-by-step UDS scheme has been introduced in searching for a feasible solution. With the intention of achieving the maximum possible search efficiency, we have taken a number of measures toward a constructive integration of CSP techniques, rule-based reasoning and ATMS techniques in our system. These techniques are used for data pre-processing, solution searching and efficient UDS, respectively. At this stage, the performance of the system is quite satisfactory. Further development of this solver may find application in practice. The method used for searching for another solution based on the user's request cannot always guarantee to carry as many as possible partial results from one solution to a new solution. In the next chapter, we use another strategy to remedy this and apply it to a crossword puzzle problem.
Chapter 7

Case Study II: Crossword Puzzle Problem

7.1 Introduction

To show how to implant both CSP techniques and the particular strategies in UDS, we give another case study on a typical example — a crossword puzzle problem. The problem posed by a crossword puzzle is to put suitable words into empty places according to certain requirements. Solving such a problem can be achieved under our UDS scheme.

In our UDS system, some constraints or some values assigned to the variables can be changed during the solution search at the request of the user. This process involves some complexity, because even a minor change in the constraints or assigned values may well make some previous search results useless. Therefore the search efficiency in UDS is strongly dependent on the capability to transfer information from previous search to further search. This requires that we find an appropriate way to store and to process the ‘knowledge’ obtained in the course of previous searching and, then, to use it for further searching.

In solving a crossword puzzle problem, constraints can also be classified into two kinds: static constraints and dynamic constraints. Static constraints describe general restrictions which apply for all the cases under consideration, while dynamic
Chapter 7: Case Study II: Crossword Puzzle Problem

Constraints represent special conditions which can be changed and modified from case to case at the user’s request. For example, “The third letter of word 1 across is the same as the first letter of word 2 down” may be a basic requirement in a crossword puzzle problem, and “The word ‘large’ cannot be used to fill the word 1” may be an additional constraint for a given requirement. Note that all the solutions for the given problem must satisfy all the static constraints and certain of the additional constraints required by the user.

In many practical applications where UDS can be used, there is a close proximity between the solutions obtained which are considered as unsatisfactory and the user-expected solutions to be obtained. This is because, generally, only minor adjustments, such as minor changes of constraints, are introduced to the next solution search. In these cases, therefore, the achievement of maximum possible efficiency is largely dependent on making effective use of the solution obtained in the previous search. Taking this into consideration, we have studied some techniques in interactive problem solving.

UDS is introduced to make the procedure of solution searching more user-friendly, efficient and controllable. Whenever it is considered necessary, we can use UDS to conveniently direct the whole process of searching.

Firstly, we carry out filtering on variable domains using our CSP module, developed in Chapter 4, in order to eliminate domain values which will not be any part of the possible solutions. These values do not satisfy certain constraints and thus will lead to useless backtracking. Our example will show that it can be very beneficial to use the CSP techniques to filter domain values in solving some AI tasks.

Secondly, to allow for general UDS, we have implanted one of the strategies presented in Chapter 5, and have classified variables into different groups. Supposing the variable set is in the form \{ v_1, v_2, \ldots, v_n \}, we partition all the variables into four groups where each is a subset of the variable set. These are named hold, base, free, and change groups respectively. The variables in the hold group should not have their assignments changed. The variables in the base group will be kept unchanged if possible, as finding a new solution is achieved by having the fewest
possible members of the base group changed. The variables in the free group are free to range over their domains in the usual way. The change group variables must be changed, that is, they cannot bring all their current values into a new solution. These classifications make user-directed search more flexible, and hence they provide a useful and effective way in which the feasible solution search can be conveniently monitored and directed.

Thirdly, we present methods for most efficiently transferring the information and results obtained into further solution search with the intention of achieving efficiency in realizing UDS. This general work on crossword puzzle problem suggests that the method has important practical applications as well.

7.2 Problem Description and Knowledge Representation

We consider a crossword puzzle problem where we are to find suitable words from the given word list to fill empty places, with duplicates allowed. For the puzzle shown in Figure 7.1, Word 1 corresponds to 1 across requiring five letters, and Word 2 corresponds to 2 down requiring five letters also. A binary constraint arises when two words intersect. That is, if two words intersect in a place, they must use the same letter in that place. For instance, the constraint between word 1 and word 2 is that the third letter of word 1 must be the same as the first letter of word 2.

We use five variables \( v_1, v_2, v_3, v_4, v_5 \) to represent five words in Figure 7.1. The domain \( D_i \) of variable \( v_i \) is all the words in word list \( (wl) \). This problem is equivalent to determining the truth value of a well-formed formula in first-order predicate logic:

\[
(\exists v_1)(\exists v_2)(\exists v_3)(\exists v_4)(\exists v_5)(v_1 \in wl)(v_2 \in wl)(v_3 \in wl)(v_4 \in wl)(v_5 \in wl)
\]

\[
P_1(v_1) \land \cdots \land P_5(v_5)P_{12}(v_1, v_2) \land P_{13}(v_1, v_3) \land P_{21}(v_2, v_1) \land P_{24}(v_2, v_4)
\]

\[
\land P_{25}(v_2, v_5) \land P_{31}(v_3, v_1) \land P_{34}(v_3, v_4) \land P_{35}(v_3, v_5) \land P_{42}(v_4, v_2)
\]

\[
\land P_{43}(v_4, v_3) \land P_{52}(v_5, v_2) \land P_{53}(v_5, v_3)
\]

where \( P_i \) is a unary constraint specifying the word length. \( P_{ij} \) denotes a binary
constraint between two intersecting words i and j.

A convenient problem representation for a crossword puzzle is a network which is a graph with a node for each variable with its associated domain attached and with a directed arc between two constrained variables. The network for the crossword puzzle in Figure 7.1 is shown in Figure 7.2, where the initial domain of words for each variable is the word list.

We create each node as a unit which is a member of NODES (created in Chapter 4) and each arc as a unit which is a member of ARCS (created in Chapter 4). The NODES unit contains slots, such as domain, and predicate, which are used for storing attributes of each node. The ARCS unit includes slots, such as binary-
7.3. Searching Scheme

A predicate, and two-nodes (see in Chapter 4). Each node unit or arc unit inherits specific information and functionality from the NODES and the ARCS respectively. Therefore, the domain and the unary predicate of each variable are stored in the corresponding node unit while the binary constraints and the pair of nodes of each arc are stored in the corresponding arc unit. We create a new slot named value in the NODES unit to keep the value assigned to a node during the course of searching. That is, the assigned value is stored in the slot named 'value' of each unit corresponding to the node unit once it is deduced during reasoning.

We also create a unit called assist which keeps the local search state on the node.list and satisfied slots of the assist unit. At any point in the search, the node.list slot contains a list of the nodes that have not yet been filled. We mark the satisfied slot of the assist unit as yes when all nodes have values that satisfy all the constraints as well as the user's request.

At the beginning of the search, the node.list slot contains a list of the nodes which need to be filled with a word. We initialize the satisfied slot to be no.

7.3 Searching Scheme

The basic search for solving the crossword puzzle is done via forward chaining in KEE. A solution is an assignment of particular values to all the variables subject to a set of constraints. The effective use of constraints is important in guiding and controlling search for a possible solution. Here we discuss how to represent constraints for our problem.

7.3.1 Deduction Rules for Constraints

We use deduction rules which cause the creation of ATMS justifications to represent constraints. When a deduction rule is instantiated, a justification will be created. The justifications ensure that, whenever all the premises of the rules match the facts, the system believes the corresponding conclusions. If a deduction rule, for example, $z \leftarrow x, y$, applies, the ATMS will build a structure — the justification. This enables
the system to recognize that, whenever we come to believe both \( x \) and \( y \) in a world, we also believe \( z \) in that world.

We use the special proposition \texttt{false} as the conclusion of a deduction rule. This kind of deduction rule has the following form:

\[
(\text{while } (\text{premise}_1)(\text{premise}_2) \cdots (\text{premise}_n) \\
\text{believe false})
\]

If such rules apply in a world, that world is contradictory and is considered \texttt{nogood}. Nogood worlds appear with solid black squares in the center (details in Chapter 2). The rule system ignores ‘nogood’ worlds in choosing rules to apply.

Deduction rules are useful to express constraints. For example, we express the constraint between \texttt{nodel} and \texttt{node2} as follows:

\[
(\text{Node1} . \text{node2} . \text{rule} \\
(\text{while} \\
\text{(the value of nodel is ?v1)}) \\
\text{(the value of node2 is ?v2)}) \\
\text{(not (equal (aref ?v1 2) \\
\text{(aref ?v2 0))})} \\
\text{believe false})
\]

where ‘aref’ is a standard function in Lisp. \texttt{(aref ?v1 2)} returns the third letter of word \texttt {?v1}. This rule means “if there is a world in which the the third letter of value in nodel is not the same as the first letter of value in node2, don’t do anything more with that world; it’s no good.” The rule system will never consider an inconsistent world. Nogood worlds present the states that violate the constraints.

Applying the above rule with a particular set of bindings for \texttt {?v1}, and \texttt {?v2} (say, ‘sound’ and ‘steer’) creates the following justification:

\[
\text{Justification:} \\
(\text{while } (\text{the value of nodel is ‘sound’}) \\
\text{and } (\text{the value of node2 is ‘steer’}) \\
\text{believe false})
\]
7.3. Searching Scheme

After thus justification is created, any world in which (the value of node1 is 'sound') and (the value of node2 is 'steer') are true will become inconsistent (nogood). The system will not consider any nogood world in the further search.

7.3.2 Action Rules for Search

KEE provides several powerful mechanisms for problem solving including a rule system which supports the use of two types of rules: deduction rules and action rules. We have already seen examples of deduction rules (see above section) and have expressed constraints as deduction rules to perform constraint checking. There are quite a number of differences between deduction rules and action rules. Deduction rules create justifications for the facts. Action rules, on the other hand, cause an action to be taken. For deduction rules, the truth of the conclusions depends on the truth of the premises. For action rules, only the application of the rule depends on the truth of the premises. After the rule has been applied, the premise become false and the conclusions will remain true.

The following is an action rule for assigning a value to a node.

(assign.node.value
  (if (the node.list of assist is ?nodes)
    (not (equal ?nodes nil))
    (equal ?first (car ?nodes))
    (equal ?rest (cdr ?nodes))
    (a domain of ?first is ?l)
  then in.new.world
    (delete (the node.list of assist is ?nodes))
    (the node.list of assist is ?rest)
    (delete (a domain of assist is ?l))
    (change.to (the value of ?first is ?l)
      using
      constraints.rules))

where a question mark followed by an arbitrary combination of letter, such as ?nodes,
?first, is a variable of the rule. These variables find their bindings when the rule's premises are tested.

To know the behaviour of the above rule requires an understanding of the scheduling algorithm of the rule system. A rule system cycles through a three-step process of:

- determining which instantiations of rules are eligible to tickle;
- selecting a particular instantiated rule to fire; and
- taking the actions required by performing that rule’s conclusions.

The rule system tests the premises of each rule to determine if all the premises of the rule are satisfied. Assume the value of the node.list slot of assist is (node1 node2 node3). All the premises in assign.node.value rule match the fact in a context. The variable ?nodes is bound to (node1 node2 node3), ?first to node1, ?rest to (node2 node3), and ?l to ‘sound’ which is one of the values of the domain slot of node1 unit. These bindings come from the matching of the premises and the facts in a world. Actually, the process of matching is to select the nodes one by one in the node.list of assist of the current world, to find a candidate domain value in the node domain.

Once all the premises of the assign.node.value rule are satisfied, this new world action rule is applied to create a new world (new state in searching) and the following actions are taken:

1. set the node.list to the rest of the previous node.list, that is, remove the first node from node.list.
2. remove the value (bound to ?l) from the domain of the node.
3. assign this value to that node, i.e. set this value as the value of the node1.

Then invoke the constraint rules. If this assignment comes into conflict with the constraints, the world is made nogood. Backtrack then occurs.

If the node.list of the assist is empty, that is, all the nodes find the corresponding values, the Ask.user rule will be tickled. This rule offers an interface so that the user
7.3. Searching Scheme

If the user is satisfied with the solution obtained, the satisfied slot of assist unit will be marked as yes. The system then exits and reports the current solution. The search process is shown in Figure 7.3. The dead end in the search tree is shown as a solid black square in the world. Each world corresponds to a decision point in the search.

The system contains both action rules and constraint rules (shown in Figure 7.4). All the rules are organized in a hierarchy which can efficiently use all the rules or only rules from specialized class, depending on the problem at hand. A well-structured hierarchy of rules narrows the search space. If we do not organize rules into a hierarchy, all the rules will be invoked. Thus, the premises of all the rules are checked to match with the facts in the knowledge base. This process will consume considerable CPU time depending on the size of a problem. In order to save CPU
time, it is important to limit the rules for consideration. In our application, we partition rules into assign.rules and constraints.rules. assign.rules is to assign a value to a node while constraints.rules is to perform constraint checking. One or both groups of the rules can be called according to the problem to be solved.

Also, it is noted that the order of premises in rules may influence the efficiency of our system. Premises are evaluated in an order given by the rule. For each premise, the system first establishes all possible bindings and then evaluates them. If the premise fails, the system will not evaluate the premises after it. So it is beneficial to place premises which are more likely to fail before premises which are less likely to fail.

Furthermore, we can invoke forward chaining on a different rule class from within forward chaining. In the above assign.node.value rule, one of the conclusions is (change.to (the value of ?first is ?l) using constraints.rules). If such a rule is applied with ?first bound to node1 and ?l bound to ‘sound’, the facts, (the value of node1 is ‘sound’) will be added in a new world. Also, additional forward chaining will be carried out using only the rules in the constraints.rules rule class, which is used for checking whether the facts asserted violate the constraints or not.
7.4 User-Directed Search Implementation

The user-directed search mechanism is introduced in order to converge upon a solution which satisfies the user. Suppose we have found one possible solution which satisfies the requirements at the initial stage. UDS allows the user to intervene and direct the process of search when the user is not satisfied with the obtained solution. This UDS process is continued until the acceptable solution is found if it actually exists. In this scheme, we need to study the methods of most efficient use of information and results obtained for further search. This means that maximum information is to be transferred from the previous solution to the next solution. Taking this into consideration, we present here another method to realise UDS, of interactive problem solving.

7.4.1 Variable Classifying

In Chapter 6, we have applied ATMS to achieve backjumping to the highest level in the search tree which contributes to the conflict with the user's new requirements and to continue further solution searching. The disadvantage of this method is that the new solution cannot always guarantee to keep as many as possible of the partial results of initially obtained solution. In the case where user-directed search is required, special investigations must be made regarding how to make efficient use of the results consistent with the current requirements in finding another solution. This is particularly beneficial in exploring the solution space to find a new solution by changing as few as possible of the results of the obtained solution.

It is assumed that S1 is an obtained solution and the constraint set for S1 is R. If S1 is unsatisfactory and an additional requirement is added by the user, another solution S2 is to be found which is required to satisfy the new requirements Φ as well. The constraint set for the solution S2 is then R1 = R ∪ Φ. Note that R and R1 are probably large constraint sets that differ in only a few constraints. Since we consider constraint strengthening UDS, R1 is stronger than R. The results obtained during the search for solution S1 can be used to make the search for a new solution S2 more efficient. For instance, we find a possible solution for the crossword puzzle
problem (shown in Figure 7.1), namely, 'sound', 'units', 'duels', 'wise' and 'hoses' for each node respectively. UDS is required when the solution is unsatisfactory. The user may advise the system the following additional conditions in its searching for another solution:

- 'sound' assigned for node1 and 'wise' assigned for node4 are accepted – keep them unchanged.

- 'duels' assigned for node3 is acceptable and keep unchanged if possible.

- 'units' assigned for node2 is acceptable and free to be changed.

- 'hoses' assigned for node5 is unacceptable and must be changed.

Our main goal is to develop a generic AI architecture for flexible exploration of the solution space, so that efficiency can be gained in searching for a new feasible solution satisfying the user’s requirements by carrying as many as possible of the results obtained in solution S1 into another solution. We create an environment in which the search can be controlled and the user-desired solution can be obtained interactively by a series of the user’s requirements. Thus, the search process could be effectively guided according to the user’s preference. Such a process is suitable for application in multiple domains.

To achieve general UDS, we have classified variables into different groups. We have handled different groups in different ways. We do the following:

Suppose we are given a solution which is an assignment of values $x_i$ to all the variables $V = \{v_1, v_2, \ldots, v_n\}$ such that all the static constraints are satisfied. We create groups $\{g_1, g_2, g_3, g_4\}$ consisting of subsets of the variables in the problem under consideration. The four groups are:

1. Hold group, $H = \{v_{h_1}, v_{h_2}, \ldots, v_{h_m}\}$
2. Base group, $B = \{v_{b_1}, v_{b_2}, \ldots, v_{b_n}\}$
3. Free group, $F = \{v_{f_1}, v_{f_2}, \ldots, v_{f_n}\}$
4. Change group, $C = \{v_{c_1}, v_{c_2}, \ldots, v_{c_n}\}$
where the union of four groups $H \cup B \cup F \cup C$ is equal to $V$, and the intersection of any two groups, such as $H \cap C$, is empty. The variables in the hold group should not have their assignments changed. The base group variables should only be changed if it is imperative to find a new solution. In other words, we keep as many as possible of the assignments in the base group unchanged. And even then, a new solution is adjacent to the solution obtained, measured by changing the smallest number of such group variables. The variables in the free group are free to be changed, ranging over their domains in the normal way. The variables in the change group are free except that they cannot take on their current values in the new solution. In other words, the change group variables must be changed. These classifications make UDS more flexible, and hence they provide a very useful and friendly environment in which feasible solution searching can be conveniently monitored and directed. The classifications of variables are decided by the user through an interface to the system.

Clearly, CSP can be used to restrict the domains of the search for a new solution since the variables in the hold and change groups have new domains. We can carry out data processing using a CSP module before starting to search for another solution. The variables in the hold group contain only one value in each domain. The number of domain values decreases by one in the change group because the variables in such a group cannot take the values in the initial solution $S_1$. Thus, CSP can be used for filtering all the variable domains of base, free, and change groups (Details are discussed in the following section).

### 7.4.2 Domain Reordering

Assume that we have selected a variable to be instantiated next. How do we select among its possible values? It is important to select a value that is the same as the one that appeared in the obtained solution. One consideration is that we can try to keep as much as possible of the obtained solution in a new solution. Another is that we select the value for the variable that it is most likely to succeed. Therefore, we reorder the domain values of each variable when a solution is obtained, and then we
can assign the value for each variable which is most likely to lead to success when re-searching for those variables which are needed. For instance, assume \( v_i \leftarrow d_j \), \( d_j \in D_i \), and \( d_j \in S_1 \). Reordering the domain values in \( v_i \) means arranging the order of values in \( v_i \) such that \( d_j \) is in the first place of \( v_i \) domain. Thus, if \( v_i \) is assigned, the value \( d_j \) is taken for \( v_i \) before all other values in \( v_i \). As a result, efficiency will be improved by reordering variable domains, that is, by first trying results obtained.

Thus, when a solution is obtained, all the variables are reordered. The values which appeared in the obtained solution will be arranged in the first place in corresponding variable domains. Figure 7.5 illustrates the properties of the reordering domains in problem solving. Figure 7.5 (a) shows the search tree of a solution. We reorder all the domains so that the values in the solution can be used prior to other values in further search. Figure 7.5 (b) shows the search tree of searching for a solution once again on the same input after reordering domains. It illustrates a variable instantiated by trying the value most likely to succeed, and it results in a better performance.
7.4. User-Directed Search Implementation

7.4.3 Variable Reordering

Generally, selecting a different order can result in large differences in the search space size. Here we do not discuss how to arrange the order of variables instantiated at the initial stage (for detail see reference [Dechter 88]). Our interest is to reorder the variables after obtaining a possible solution according to the classifications of variables.

We can re-arrange the order of variables in order to achieve searching efficiency. Before searching for a new solution, the user gives information about variable classifications, so that variables can be ordered according to these data. The variables in the hold group should not have their assignments changed. Thus a new solution will keep these parts. The variables in the base group will be changed as little as possible. We discuss this group in the next subsection. The variables in other two groups free and change are arranged like:

\[
(v_f v_f \cdot \cdot \cdot v_{f_n} v_c v_{c_2} \cdot \cdot \cdot v_{c_m})
\]

\[
\text{free group } \quad \text{change group}
\]

In the assign.node.value rule, we select the first node in the node.list slot of assist. Then we remove it from node.list. The second node becomes the first one in node.list. Thus, the order of nodes to be filled is completely dependent on the order of nodes in node.list. We start new solution searching by creating a world start. We make a list of the above variables, as the value of the node.list of assist in world start.

We have already reordered the domain values for each variable, so that in the assignment of each variable we can first try the one which appeared in the solution obtained except in the case of the changed domains, such as the variables in the change group.

During solution searching, we exploit action rules for variable assignments, deduction rules for constraint checking, and worlds for expressing the problem-solving state, which can be used for preserving the search states.
7.4.4 Minimum Changes

In some cases, a possible solution cannot be found, even when we try all the assignments of variables in the free and the change groups. In this case, we have to change the variable assignments in the base group. Our main aim is to change the smallest number assignments of the base group variables.

Now we discuss how to deal with proximal minimum changes of the variables in the base group \( B = \{ v_{b_1}, v_{b_2}, \ldots, v_{b_{n_2}} \} \) for new solution searching. If each variable has a weight for deciding the order for change, we can arrange the order of variables in the base group according to their weights.

Since we change the smallest number of variables in the base group, we first consider changing only one variable at a time. When the variable value is revised, an assignment of this variable can range over all the values in its domain except the current value. And the remaining variables in the base group keep their values unchanged.

This is shown in Figure 7.6, where 1 stands for the variable that takes the same value as is does in the initial solution \( S_1 \), and 0 means that the corresponding variable assignment is changed, that is, it takes any value except the value in solution \( S_1 \). In other words, ‘1’ means unchanged while ‘0’ means changed. The number of cases where only one variable is changed is \( C^1_{n_2} = n_2 \).

Figure 7.7 shows the case of ‘two variables changed’, including the variable \( v_{b_1} \). Actually, consideration should be taken of any pair of variables in the base group. The total number of cases where two variables are revised equals \( C^2_{n_2} \).

In the worst case, all the variables in the base group have to be revised. That is,
7.5. User-Directed Search Implementation

Figure 7.7: The cases where variable $v_{b_1}$ and any one other variable are revised.

$$\begin{array}{cccc}
0 & 0 & 1 & \cdots & 1 \\
0 & 1 & 0 & \cdots & 1 \\
0 & 1 & 1 & \cdots & 1 \\
\vdots \\
0 & 1 & 1 & \cdots & 0 \\
\end{array}$$

Figure 7.8: Case where all the variable are changed.

$$\begin{array}{cccc}
0 & 0 & 0 & \cdots & 0 \\
\end{array}$$

they can take any values in the corresponding domain except the values in solution S1. Figure 7.8 shows that the values of all the variables in the base group are changed.

Our motivation is to change the minimum number of variables in the base group. Suppose $v_{b_1}, \ldots, v_{b_{n_2}}$ are in the base group and their corresponding values in solution S1 are $a_1, \ldots, a_{n_2}$. An informal description of the minimum change algorithm, which can easily be translated into any AI language, can be found in section 5.5.

In Figure 7.9, $a_1, \ldots, a_{n_2}$ are assigned to variables $v_{b_1}, \ldots, v_{b_{n_2}}$ in the base group respectively. And $l_{1i} \in D_{b_1} - \{a_1\}, 1 <= i <= t_1$ ( $t_1$ is the number of values in domain $D_{b_i}$). The difference of the base group variables increases from Level 1 to Level $n_2$. In Level 1, only one variable assignment is revised.

We create a world as a child of world start to represent a change on the base group. Then we invoke the constraint rules to check whether or not this change conflicts with the constraints. If it does conflict, the world becomes nogood and it is ignored for further search. Otherwise, we invoke Assign.rules to search for a new solution according to the node.list. If this fails to produce a possible solution, once again the base group needs to be changed. In this way, we can search for a new solution based on minimum change of the base group variables and can monitor the solution search by using UDS.
The crossword puzzle can be presented as a typical constraint satisfaction problem. Thus we can use CSP techniques for deleting domain values which cannot be part of any possible solution during the solution searching in a UDS. CSP techniques can be employed to perform data processing prior to the solution search and during the course of problem solving. This process can be called data processing or filtering.

### 7.5.1 Data Pre-Processing

Pre-processing usually reduces the original set of the variable values down to a subset. Then the search will be carried out on the smaller space. We use the same example described previously. Figure 7.1 shows a crossword puzzle problem in which any word can be chosen from the word list. We create the constraint network for it, as shown in Figure 7.2. Each initial domain contains 24 words. If we do not perform pre-processing, the number of different assignments to be tested is:
where \( N_i \) is the number of values (words) in each variable domain and \( n \) is the number of nodes. Here \( N \) equal to 24 and \( n \) is 5. Thus

\[
\Pi_{i=1}^{n} N_i = 24^5
\]

After carrying out data pre-processing, five variable domains are reduced to the following:

- \( D_1 = \{ \text{hoses round sound world} \} = 4 \) values;
- \( D_2 = \{ \text{sails round sound slots units} \} = 5 \) values;
- \( D_3 = \{ \text{duels slots} \} = 2 \) values;
- \( D_4 = \{ \text{wise auto line rule} \} = 4 \) values;
- \( D_5 = \{ \text{laser nodes hoses} \} = 3 \) values.

After performing data pre-processing, the number of different assignments to be tested now is

\[
\Pi_{i=1}^{n} N_i = 4 \times 5 \times 2 \times 4 \times 3.
\]

These results clearly show that the total search space is reduced by performing data preprocessing.

By carrying out data pre-processing prior to solution search and using the rule system in KEE, we can obtain a possible solution when we utilize the rule system in KEE to search, as shown in Figure 7.10.

### 7.5.2 Two Level Filters in UDS

In order to improve solution search efficiency in a UDS, we can also exploit CSP techniques to filter domain values each time after the user makes a new classification of variables. This is because the variable domains in the hold and change groups are
Chapter 7: Case Study II: Crossword Puzzle Problem

Word List

<table>
<thead>
<tr>
<th>s</th>
<th>o</th>
<th>u</th>
<th>n</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>u</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w</td>
<td>i</td>
<td>s</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>t</td>
<td>l</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- hoses sails duels well drink
- round sound wise auto line
- nodes laser world units slots
- rule false jenny god fact
- day are game done

Figure 7.10: A possible solution of the crossword puzzle in Figure 7.1.

changed with the new classification. Thus, efficiency can be gained by using CSP techniques to reduce variable domains and to yield a smaller search space.

Suppose we have found a possible solution $S_1$, as shown in Figure 7.10. The user rejects it, and a new classification of variables is given through user interface as follows:

- Hold group: $v_5$
- Base group: $v_3$
- Free group: $v_1, v_3$
- Change group: $v_4$

The variables in the hold group should not have their assignments changed, so these variable domains are changed to contain only one value. That is, $D_5 = \text{hoses}$. The variables in the change group must be changed, i.e. they cannot take on any of their current values in a new solution. Thus, these variable domains are changed by eliminating one value which is in the solution $S_1$. The domain of $v_4$ in the change group is changed from $\{\text{wise auto line rule}\} = 4$ values to $\{\text{auto line rule}\} = 3$ values.

If domain $D_i$ of $v_i$ is changed, all the arcs which lead to this variable $v_i$ in the constraint network require arc consistency checking. In the above example, $D_4$ and $D_5$ are changed. After applying the CSP module (developed in Chapter 4), some of
the domain values of $D_1, D_2, D_3$ may be eliminated. The resulting domains, after carrying out the CSP module, are as follows:

- $D_1 = \{\text{hoses round sound}\} = 3$ values;
- $D_2 = \{\text{sails slots units}\} = 3$ values;
- $D_3 = \{\text{duels slots}\} = 2$ values;
- $D_1$ and $D_2$ are reduced. We consider the above process as the first level filter. This process is carried out right after a new classification of variables is given.

The second level filter occurs when the variables in the base group are instantiated. In the above example, if the variable $v_3$ in the base group is assigned a value $d_3i$, we modify the domain of $v_3$ containing only one value $d_3i$. Then we use this information to reduce other variable domains. In this way, search efficiency for a new solution can be improved by eliminating domain values which will not appear in any possible solution.

### 7.6 An Example of UDS

When the system has found a possible solution which satisfies a set of static constraints, this solution will appear on the user's screen (see Figure 7.11(b)) for the

---

**Figure 7.11:** A search space within user-directed search.

---
user to decide whether it is satisfactory or not. If it is, we exit with the current solution. Otherwise, menus will be brought up on the screen to enable the user to classify variable groups, if considered necessary. Assume the user gives the following information:

- Hold Group: Node5
- Base Group: Node3 Node4
- Free Group: Node1
- Change Group: Node2

This means that a new solution must satisfy the following conditions:

1. The assignment of Node5 in the hold group should not be changed;

2. Keep, if possible, the assignments of Node3 and Node4 in the base group unchanged;

3. The assignment of Node1 in the free group is free and can take any value in the domain;

4. The assignment of Node2 in the change group should be changed. That is, it cannot take on its current value in the new solution.

The new solution must satisfy the above-mentioned requirements in addition to static constraints. The search space for the new solution is shown in Figure 7.11(a). The search begins from the world start, which contains initial knowledge and the variable assignments in the hold group. The direction of search is from top to bottom, and from right to left. Each world corresponds to a decision point in the search. The dead end in the search tree is shown as a solid black square in the world. The process of searching is as follows:

First, keep the variable assignments in the base group unchanged. Based on partial results (the variable assignments in the hold and base groups are the same as the solution obtained), search the rest to find the new solution. If the search fails, variable assignments in the base group have to be changed. The second level in the search space in Figure 7.11 expresses the changes in the base group. If the change of
variable assignments in the base group conflicts with static constraints, the system makes it nogood. As shown in Figure 7.11(a), the worlds 'base.group.changed25' and 'base.group.changed26' are inconsistent, thus no further searching follows from them. The new solution is shown in Figure 7.11(c).

The new solution can be searched based on the considerations and techniques developed earlier and it is close to the obtained solution. If the user accepts the solution, the system reports it and finishes its task. If the user rejects it, a step-by-step UDS will continue until an acceptable solution is found, if it exists. The user can input different classifications of variables. This makes the procedure of solution searching more friendly, efficient and controllable.

7.7 Summary

The ability to allow the user to intervene and direct the process of searching and to adapt to a changing environment is an important aspect of intelligent behaviour in expert systems. For some practical applications, there is a great similarity among the possible solutions to a given problem. For such cases, efficiency can be gained by finding a possible solution and carrying as many as possible of the obtained results into another solution. We have established the user-directed search (UDS) mechanism to gain efficiency in moving from one solution to other solution.

With the intention of searching for the closest solution to the solution obtained, we have presented the method for most efficiently transferring the information and results obtained into further solution search to satisfy the user's new requirements. To allow for general considerations of finding the closest solution to the solution obtained, we have classified variables into different groups, namely, hold, base, free and change. These classifications make UDS more flexible.

We have shown how to exploit the CSP module (developed in Chapter 4) for data pre-processing in advance of searching and filtering domain values in UDS. Thus, those domain values which will not appear in any possible solution can be eliminated.

At this stage, the performance of the system is quite satisfactory. Importantly,
the techniques developed here, which enable quite different compatible techniques to optimize the efficiency of solution searching, can be very useful in improving the performance of some practical AI systems. Further development of this solver may find practical applications.
Chapter 8

Conclusions and Further Directions

In this thesis, we have studied some fundamental problems with regard to establishing interactive problem solving systems.

To establish such systems, we have introduced a new search scheme which we have called the user-directed search (UDS). We know that usually there can be a number of possible solutions to a particular problem under given conditions. In many practical cases, however, only one feasible solution is sufficient for a given problem. The acceptance or rejection of a possible solution from the problem solver will be determined by the user. For these cases, naturally, a user-desired solution can be explored most efficiently if a ‘communication’ mechanism is introduced between the user and system. This mechanism enables a user to guide the solution searching in his most preferred directions and thus the system can then find the user-desired solution in an efficient way. In our UDS, the interaction mechanism is introduced in such a way that the user can intervene in and control the process of searching so that the search will converge to a solution which satisfies the user.

We have used two different approaches to realize UDS.

In the first approach, we have established in the user interface an environment which enables a user to specify his changes to the initial conditions, or to add his new requirements, in the form of deduction rules. In this approach, deduction rules
can be created upon the user's request in practical environments.

We have taken a strategy to maximize the search efficiency in this approach. This is the backjump search which can first find, and then backjump to, the point which contradicts the user's new conditions or requirements. To implement this strategy, we have used techniques in truth maintenance systems (TMS) and KEE-worlds for maintaining and utilizing the information from the previous solution search. We have created some special deduction rules which can be used to create corresponding TMS justifications. These rules are used to represent generalized dependencies between facts and thus to control the search process. Using these rules, any subsequent solution search can take advantage of the information obtained in the previous search. Thus this scheme, in terms of search efficiency, can be better than that restarting the problem solver and searches for a new solution from the very beginning.

In the second approach, we have introduced some particular variable groups. These groups have been set to have different priorities in the search. In this approach, the user's requirements are introduced to further solution search through a scheme whereby the user divides the variable set into groups. By organizing or arranging these variable groups according to his preference, a user can effectively control and instruct the system in the whole course of problem solving.

In this approach, we have implemented a scheme which is called proximal (closeness) change. The proximal change ensures that, in the direction specified by the user, the closest solution to that previously obtained will be found if it actually exists. Using this scheme, we can effectively make use of the information accumulated in the previous search to seek the solution in the user's desired direction. Again, the search efficiency can be improved by not abandoning all the information obtained and not restarting the search from the beginning.

In another aspect, we have studied the special features of CSP techniques and have applied them to some different systems to improve solution search efficiency. In particular, we have studied how to apply CSP techniques in establishing expert systems such as rule-based and frame-based systems on KEE. In fact, we find that
CSP techniques can be very useful in improving search efficiency in these systems, when they are applied to perform consistency checking prior to searching for a solution, that is, ‘pre-processing’.

We have introduced the pre-processing step to eliminate a number of the candidate values of variables which will be inconsistent with corresponding unary and binary constraints. A particular set of constraints is set up and used for preparing a reduced search space to start a solution search. As a result, this step can be used to avoid a certain amount of useless backtrack searching.

We have developed an independent module particularly for applying CSP techniques in general purpose programming in KEE. Although KEE is a very powerful artificial intelligence environment which provides a number of programming tools and techniques for establishing rule-based and frame-based expert systems, it does not provide any access to CSP techniques. The implementation of such a CSP module provides KEE with improved ability to establish more versatile expert systems.

In our case studies of truck dispatching and word puzzle problems, we have demonstrated that CSP techniques can effectively reduce certain useless backtracking. Moreover, from these case studies, we have shown how to achieve UDS and how to implement various techniques which we have developed to realize a UDS. From these case studies, the fundamental considerations and usefulness of UDS can be seen clearly.

The research of this thesis can be continued in several directions. For example, the CSP module could be further developed to include some particular techniques for the dynamic constraint satisfaction problem (DCSP). In practical applications, many expert systems operate in dynamic environments where requirements and conditions, and accordingly variables and constraints in these systems, may change rapidly. DCSP techniques allow one to perform consistency checking on a dynamic range of variables and constraints. With the extension of including DCSP techniques, the CSP module would be more useful and powerful.

Also, an important and interesting question remains: how to allow a UDS to most efficiently answer queries which involve constraint relaxation. In practical applica-
tions, some problems may be overconstrained and no acceptable solution, or even no possible solution, will be found by the systems. When these situations occur, a user may have to change some requirements so that certain constraints are relaxed, and try to launch another search. In this regard, some specific methods and techniques, which could most efficiently make use of the previous search in further search with certain constraints relaxed, could be developed. Further, different user's interfaces and control strategies to allow UDS to be implemented in various expert systems and in different programming environments are yet to be developed. Also, further work will touch on the need for user assistance in moving around in solution spaces. We believe that continuing investigations in these directions will make the potential of the UDS in the development of interactive expert systems more apparent.
Appendix

1. The truck dispatching rules are as follows:

Ask.user  When a possible solution is obtained, this rule is invoked and sends messages to create menus and to ask the user for information.

\[
\text{if (the trip.list of assist is nil)} \\
\text{then} \\
\text{ (lisp (unitmsg 'assist 'ask.user.fn)))}
\]

Assign.driver  A trip is assigned a driver.

\[
\text{if (the trip.list of assist is (list.of (?first . ?rest)))} \\
\text{ (a pending.trip of assist is ?first)} \\
\text{ (not (equal ?first nil))} \\
\text{ (a candidate.driver of assist is ?d)} \\
\text{ (cant.find (the trip.nogood.driver of ?first is ?d))} \\
\text{ (the truck of ?first is ?truck)} \\
\text{ (cant.find (the truck.nogood.driver of ?truck is ?d))} \\
\text{then} \\
\text{ in.new.world} \\
\text{ (delete (a candidate.driver of assist is ?d))} \\
\text{ (change.to (the driver of ?first is ?d) using no.rules)} \\
\text{ (lisp (assert nil 'constraints.rules)}
\]
nil
:interesting.world
$world$))
(delete (a pending.trip of assist is ?first))
(delete
  (the trip.list
   of
    assist
   is
    (list.of (?first . ?rest))))
(the trip.list of assist is ?rest))

Assign.truck A trip is assigned a truck.
(if (the trip.list of assist is ?l)
  (equal ?first (car ?l))
  (not (equal ?first nil))
  (a candidate.truck of assist is ?v)
  (cant.find (the trip.nogood.truck of ?first is ?v))
then
in.new.world
  (delete (a candidate.truck of assist is ?v))
  (change.to (the truck of ?first is ?v)
    using
    'constraints.rules)
  (a pending.trip of assist is ?first))

Make.candidate.drivers Collect candidate drivers in the knowledge base.
(if (?d is in class drivers)
then
(a candidate.driver of assist is ?d))
Make.candidate.trucks *Collect candidate trucks in the knowledge base.*

(if (?v is in class trucks)
  then
  (a candidate.truck of assist is ?v))

Stop.all *Exit when an acceptable solution is obtained.*

(if (the satisfied of assist is yes)
  then
  (the problem of assist is solved)
  (lisp (prog nil
    (format t "good luck!!")
    (setq $fc.agenda$ nil))))

2. Some of the static constraint rules in the truck dispatching problem is presented as follows:

*Cant.exceed.volume.limits* *The goods carried at any time on this trip cannot exceed the volume limit of this truck.*

(while (?t is in class trips)
  (the volume.capacity of (the truck of ?t) is ?w)
  (the max.volume of ?t is ?m)
  (lisp (> ?m ?w))
  believe
  false )

*Cant.exceed.weight.limits* *The goods carried on this trip cannot exceed the weight limit of this truck.*

(while (?t is in class trips)
  (the max.weight of ?t is ?m)
(the weight.capacity of (the truck of ?t) is ?w)
(lisp (> ?m ?w))

believe
false)

Driver.and.truck.must.be.in.same.city

(if (?t is in class trips)
 (the location of (the driver of ?t) is ?d1)
 (the location of (the truck of ?t) is ?v1)
 (not (equal ?d1 ?v1))

then
(lisp
 (add.value (the truck of ?t)
 'truck.nogood.driver
 (the driver of ?t)))
(lisp (add.value ?t
 'trip.nogood.driver
 (the driver of ?t)))
(lisp (assert 'false nil $world$))

Driver.must.be.where.trip.starts

(if (?t is in class trips)
 (the origin of ?t is ?o)
 (the location of (the driver of ?t) is ?l)
 (not (equal ?o ?l))

then
(lisp (add.value ?t
 'trip.nogood.driver
 (the driver of ?t)))
(lisp (assert 'false nil $world$)))
Driver.must.have.right.licence.rule A driver must hold a right licence to drive a truck.

(while (?t is in class trips)
  (the licence.class of (the truck of ?t) is ?tc)
  (the licence.class of (the driver of ?t) is ?dc)
  (lisp (> ?tc ?dc))
  believe false)

One.trip.per.driver

(while (the driver of ?t1 is ?d)
  (the driver of ?t2 is ?d)
  (not (equal ?t1 ?t2))
  believe false)

One.trip.per.truck

(while (the truck of ?t1 is ?v)
  (the truck of ?t2 is ?v)
  (not (equal ?t1 ?t2))
  believe false)

Trip.cant.last.too.long The duration of the trip cannot be greater than the time a driver is allowed to drive.

(while (?t is in class trips)
  (the duration of ?t is ?m)
  (the maximum.driving.time of (the driver of ?t) is ?w)
  (lisp (i ?m ?w))
  believe false)
false)

Truck.must.be.where.trip.starts

(if (?t is in class trips)

  (the origin of ?t is ?I)

  (the location of (the truck of ?t) is ?vl)

  (not (equal ?vl ?I))

then

  (lisp (add.value ?t

       'trip.nogood.truck

       (the truck of ?t)))

  (lisp (assert 'false nil $world$)))
Bibliography


[Byte 86] Byte: Special Issue on Object-Oriented Languages, August 1986.


Bibliography


<table>
<thead>
<tr>
<th>Reference</th>
<th>Title and Authors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shearer 87</td>
<td>C. Shearer, <em>KEE and POPLOG: alternative approaches to developing major Knowledge based systems</em>, Presented at KBS 87, pp.79–88.</td>
</tr>
</tbody>
</table>


