CONSTRAINT-DIRECTED SEARCH IN
KNOWLEDGE-BASED PLANNING
ENVIRONMENTS

A thesis submitted for the degree of
Doctor of Philosophy
of the
Australian National University

Dharmendra Prakash Sharma
January 1993
Statement

I hereby state that this thesis contains only my own original work except where explicit reference has been made to the work of others.

\[ \text{Signature} \]

Dharmendra P. Sharma

©Dharmendra Prakash Sharma 1993
Dedication

I dedicate this thesis to my wife Madhuri, our daughter Nandita and son Shivendra. I have been fortunate to receive the understanding, patience and limitles love from the family. In addition to her sacrifices in looking after the children and allowing me to concentrate on my thesis, Madhuri has given me constant care, support and encouragement over the time of the thesis. She always stood by me in times of difficulty. Nandita and Shivendra have also been very understanding and supportive.
Machines—with their irrefutable logic, their cold preciseness of figures, their
tireless, utterly exact observations, their absolute knowledge of mathematics—
they could elaborate any idea, however simple its beginning, and reach the
conclusion. Machines had imagination of the ideal sort—the ability to con-
struct a necessary future from a present fact. But Man had imagination
of a different kind; the illogical, brilliant imagination that sees the future
result vaguely, without knowing the why, nor the how; an imagination that
outstrips the machine in its preciseness. Man might reach the conclusion
more swiftly, but the machine always reached it eventually, and always the
right conclusion. By leaps and bounds man advanced. By steady, irresistible
steps the machine marched forward.

John W. Campbell, JR.

... it is not the thing done or made which is beautiful, but the doing. If we
appreciate the thing, it is because we relive the heady freedom of making it.
Beauty is the by-product of interest and pleasure in the choice of action.

Jacob Bronowski
Acknowledgments

I am very grateful to my supervisor, Professor Robin Stanton of the Department of Computer Science, the Australian National University, for being my mentor. His comments, criticisms and insights into issues relevant to the research were of tremendous help in my understanding, clarification and focusing of the research work. Dr. Graham Williams of the CSIRO Department of Information Technology critically read earlier drafts of my thesis. He always found time to be of help and had been a constant source of encouragement during the time of the research. Numerous discussions with Seppo Keronen helped me clarify my initial thoughts on the thesis work. Hugh Mackenzie provided valuable comments. Dr. Walter Hower of Universität Koblenz-Landau gave comments, answered many queries on constraint satisfaction and provided me with valuable references. I also had useful discussions with Dr. Charles Petrie of the MCC AI Laboratory. Kerry Taylor commented on some of my writings. Dr. John Smith and Dr. Vicki Peterson helped in my understanding of the ANU BSc regulations. James Popple helped with \LaTeX. My sincere thanks to them all.

The funding came from the International Development Programme (IDP) of Australian Universities and Colleges and my home university, the University of the South Pacific (USP). I owe a great debt of thanks to the two institutions. IDP deserves a special mention for looking after my family and for all the help they have given us during our stay here in Australia. USP gave me leave to allow me to undertake the research. The Department of Computer Science at the ANU provided an excellent environment for the research to be undertaken. The members of the department made me feel at home. I thank them for their kind gesture.

I sincerely acknowledge the support and blessings of my parents who have toiled to give the best of everything to their children. This is yet another honourable achievement for them by one of their children.

Finally, a special thanks to my family for providing me unfailing support and encouragement in my seeing through probably the most challenging endeavour in one’s life.
Abstract

This thesis investigates a class of real-world problems which bring together two distinct areas of research in artificial intelligence (AI)—planning and constraint satisfaction. Virtually all planning problems involve constraint satisfaction of some kind. Classical planning problems involve search for sequences of actions which, when executed by an agent, result in some desired goal that is specified by some given set of constraints. Not all planning problems require search for fine-grained action sequences.

The class of problems investigated require search for plans that satisfy a given set of a variety of constraints. Here, the plans are sequences of sets which contain plan objects (or activities). The problems are essentially dynamic constraint satisfaction problems. Characteristically, in these problems, not all the constraints, variables and their respective domains may be known a priori; there is usually a partial ordering between different classes of constraints; the constraints may be formal (and hard) or preferential (and defeasible); constraints may be coupled (satisfaction of some constraints may result in satisfaction of other constraints); constraints express both extensional and intensional relationships; constraints may be dynamically added and removed at runtime; problem solving requires user-guided navigation and exploration of search space (through preferential constraints); constraints belong to different functional categories; constraint properties are exploited in effective constraint-directed search guidance and search reduction; and constraints may become inconsistent during problem solving.

The classical constraint satisfaction problem (CSP) model does not embody many of the above features. Also, the incremental and interactive features, as usually required in modeling real-world problem solving, are inadequately addressed in many CSP formulations. Further, the class of problems does not strictly conform to the characteristics of planning problems as accepted by the AI community albeit they are “planning” problems.

We develop a theory of Constraint-Directed Search for Planning Problems (CDSPs) for which neither classical AI planning nor constraint satisfaction formalisms are adequate. The CDSP theory supports reasoning with dynamically changing constraints. It is built on a constraint language in which CDSP problems are specified. Two types of problems and their respective problem solving frameworks are identified from the generic CDSP class of problems, and formulated. They are problem solving from “scratch” treating each new set of constraints as a new problem (as adopted in CSPs), and incremental problem solving. We formalise these types and give the theoretical underpinnings. Problem formulations, solution structures and problem solving schemes for both these types of problems form the components of the proposed theory.

Discrimination between constraints and their exploitation in guiding search are
characteristic features of the CDSP theory. User interaction in the exploration of the solution space and in guiding search is an essential dimension of the theory. The above constraint-directed reasoning features are supported. In addition, CDSP incorporates object representation of domain and control knowledge (including constraints), advice generation from existing problem solving states, and conditional query processing. Heuristics are drawn from experiential knowledge, if available, and exploited in search reduction and advice generation. The constraint language allows the user to participate in problem solving by dynamically guiding search via preferential constraints.

A novel computational architecture that implements the CDSP model is presented. It incorporates a novel application of an assumption-based truth maintenance system (ATMS) to manage reasoning with constraints is provided. A single-context ATMS is developed to capture the problem solving knowledge and facilitate further reasoning with it. It is used by a dependency-directed backtracking problem solver to adequately model CDSP problem solving. The ATMS provides semantics for traditional inconsistency detection and resolution.

Planning programmes of study for academic degrees has motivated this research. It manifests many novel and interesting research issues many of which have been addressed in this thesis. Experiments conducted in designing and developing an intelligent knowledge-based system for degree planning, culminating in the CDSP model are reported. BSc degree planning from the Australian National University has been used as the experimental domain.
Contents

Acknowledgments v

Abstract vi

1 Introduction 1
  1.1 Problem Solving Research in AI .......................... 1
  1.2 Constraints in Planning ................................ 2
  1.3 Current Research ...................................... 4
  1.4 Overview of the Research Problem ....................... 4
    1.4.1 Degree Planning .................................. 4
    1.4.2 Other CDSP Problems ............................. 6
  1.5 Primary Contributions ................................ 7
  1.6 Chapter Summaries .................................... 10

2 Previous Work 12
  2.1 Introduction ......................................... 12
  2.2 Search in Problem Solving ............................ 12
  2.3 Planning ............................................. 13
    2.3.1 Planning Techniques ............................. 14
    2.3.2 Applications ................................... 16
  2.4 Constraint Satisfaction ............................... 17
    2.4.1 Characteristics ................................ 17
    2.4.2 Search-Based Approaches ....................... 18
    2.4.3 Constraint Propagation Approaches ............. 20
  2.4.4 Constraint-Satisfaction Systems and Applications 20
    2.4.4.1 Problem-Solvers for CSPs ................... 21
    2.4.4.2 Constraint Languages ....................... 21
    2.4.4.3 Applications ............................... 22
3 CDSP Model

3.1 Introduction

3.2 Solving CDSP Problems
   3.2.1 A General Approach
   3.2.2 Tasks in CDSP Problem Solving
   3.2.3 CDSP Model: A Top-Level Description

3.3 A Characterisation of CDSP Problems
   3.3.1 Plans
   3.3.2 Plan Objects
   3.3.3 Constraints
   3.3.4 Incrementality
   3.3.5 Computational Intractability
   3.3.6 Constraint-Directed Search
   3.3.7 Constraint Propagation
   3.3.8 Dynamic Constraint Satisfaction
   3.3.9 Exploration of Solution Space
   3.3.10 Inconsistency
   3.3.11 Constraint Relaxation
   3.3.12 Plan Revision
   3.3.13 Plan Verification and Repair
   3.3.14 Data Dependency
   3.3.15 Experiential Knowledge

3.4 Existing Models
   3.4.1 CDSP Problems as Constraint Satisfaction Problems
   3.4.2 CDSP Problems as Dynamic CSPs
   3.4.3 CDSP Problems are Atypical Planning Problems
   3.4.4 CDSPs are Atypical OR Problems
   3.4.5 Constraint-Based Problem Solving Languages

3.5 Overview of CDSP Problem Solving Framework
   3.5.1 Problem Statement
   3.5.2 CDSP Problem Structure
   3.5.3 CDSP Problem Solving Language
   3.5.4 CDSP Problem Solving Architecture
D.1 Search Structure and Constraints .................................. 237
D.2 Execution of Model-Search for Degree Planning ............... 239

E Illustration of IPLAN with Degree Planning .................. 252
E.1 Constraints .......................................................... 252
E.2 IPLAN Problem Solving .............................................. 252

F Miscellaneous Algorithms and Functions ..................... 269
F.1 Miscellaneous Functions ........................................... 269
F.2 Scheduling Algorithms ............................................. 270

G Publications ............................................................ 273
List of Tables

1.1 Steps involved in experimental research in AI .................................. 3
2.1 Knowledge-Based Planning Applications. ......................................... 17
4.1 An Object Schema in ORL. ............................................................. 57
4.2 ORL Syntax .................................................................................. 58
4.3 A Plan Object in ORL. ................................................................. 59
4.4 A Constraint Object in ORL. ......................................................... 59
5.1 A Summary of Constraint Expressions in CL. ................................. 83
5.2 An XPLAN Schema. ................................................................. 86
7.1 Interface functions for CDSP-ATMS. .............................................. 181
8.1 An Example of a Course Object in ORL. ........................................ 187
8.2 A Constraint Object in ORL. ....................................................... 188
8.3 A Consistent Schedule. ............................................................... 189
8.4 An XPLAN Object. ................................................................. 190
8.5 A Case Object. ................................................................. 191
B.1 EBNF Symbols ........................................................................ 227
B.2 A Summary of the CL Vocabulary. ............................................ 232
F.1 Extend-Schedule Algorithm. ......................................................... 271
F.2 An Algorithm for Addition of a CSS to a Schedule. ................. 271
F.3 An Algorithm for Drawing a Schedule from a CSS. ............... 272
List of Figures

1.1 A degree planning system. ............................................... 5
3.1 CDSP Model. ................................................................. 28
3.2 Summary of Typical Categories of Constraints in CDSP Problems. ... 32
3.3 A Conceptual Model of the Constraint Satisfaction Metaphor. ........ 48
3.4 Supports in a CDSP Plan. .................................................. 50
3.5 Components of a General Problem Solving Language. .................. 51
3.6 CDSP Architecture. The arrows show flow of information. ............ 52
4.1 Knowledge Representation in a Reasoning System. ........................ 54
4.2 State-Based Search Structure. ............................................ 60
4.3 A SStruct Structure. ...................................................... 63
4.4 II Structure: A SStruct Combination. ................................... 64
4.5 Fission of II into Conjunctive Support Structures. ....................... 65
4.6 Addition of CSS Structures. .............................................. 68
4.7 Examples of CSS Conversions to A. .................................... 69
4.8 A Constraints on Scheduling. ............................................. 73
4.9 A Schedule Template. ..................................................... 76
4.10 Addition of a Well-Formed Temporal Structure to a Schedule. ........ 77
6.1 A State-Based Search. ..................................................... 114
6.2 A Context. ................................................................. 115
6.3 Summary of Constraint Categories. ...................................... 120
6.4 Search Space from Model-Search. ...................................... 136
6.5 Model-Search Problem-Solving Cycle. .................................. 138
7.1 Inference Engine/TMS Interaction in a Reasoning System. ............... 152
7.2 Incremental Problem Solving. .......................................... 156
7.3 A Search Space in CDSP Problem Solving. ................................ 159
7.4 An Illustration of a CDSP-ATMS Structure. ............................ 164
8.1 Information Processed in Degree Planning. ................................ 185
8.2 Degree Planning in CDSP Model. ....................................... 186
Chapter 1

Introduction

1.1 Problem Solving Research in AI

Over the last two decades extensive research effort has been expended in the development of problem solving models for computer solutions of various complex and difficult practical problems. Many of these problems involve simulation of human experts in some problem solving domain and exhibit characteristics of ill-structured \([112, 148]\) problems which cannot be solved by conventional algorithmic approaches. Artificial intelligence (AI) problem solving models have been evolving over the years. These models embed logical and heuristic reasoning, make judgments, and derive conclusions in a specific domain by drawing on encoded factual, heuristic and other domain-specific knowledge, problem-solving methods, inference procedures, and by exploiting search techniques. They usually involve symbolic information processing.

Knowledge-based (or expert) systems are knowledge-intensive automated-reasoning systems which realise these problem solving models in some narrow, specialist domains of application. Some of the highly successful knowledge-based systems developed include MYCIN (for medical consultations) \([145]\), TAXMAN (for legal reasoning) \([101]\), R1 (for configuring VAX computer systems) \([104]\), HEARSAY-II (a speech understanding system) \([50]\), ISIS (a knowledge-based system for factory scheduling) \([62]\), and MOLGEN (for planning with constraints in molecular genetics experiments) \([155]\).

A problem-solving (or computational) model forms the heart of an AI system. It is a scheme for organising reasoning steps and domain knowledge to construct a solution to a problem. Research in AI problem solving has mostly concentrated on reasoning techniques which can be employed for different problem-solving tasks. Some of these techniques are embedded in problem-languages for specific classes of tasks such as CHARME—a constraint programming language for solving scheduling and resource
alignment problems [136], REF-ARF—a system which accepts and solves problems stated in a programming language [54], ALICE—an input language to drive a system for solving combinatorial problems [92], and CONSTRAINTS—a language of hierarchical constraint networks embodying local propagation for constraint satisfaction [156]. Others are presented as general purpose problem solvers such as GPS [51,117], STRIPS [55], and blackboard architectures [123].

Much AI research in problem-solving is problem-driven (as opposed to creating solutions in search for problems) in which a researcher endeavours to find a specific problem-solving approach to solve some particular problem (or more usefully, a class of problems). The problem characteristics prescribe the approach to be adopted. When found, the approach is validated, generically abstracted, and theoretically characterised. Hence, AI research has both theoretical and experimental concerns. As Newell and Simon explained in their Turing lecture,

> Each new program that is built is an experiment. It poses a question to nature, and its behavior offers clues to an answer....[Computer Systems] are artifacts that have been designed...and we can open them up and look inside. We can relate their structure to their behavior and draw many lessons from a single experiment [116].

Furthermore, Buchanan [19] acknowledges that “expert systems offer an experimental apparatus that can enhance the scientific nature of investigations in AI”. His taxonomy of AI research into “theoretical”, “engineering” and “analytical” steps (see Table 1.1) supports the view that AI is an experimental science. GPS [115] is a good example of research meeting the characteristics outlined in Table 1.1. The present research follows this experimental paradigm.

### 1.2 Constraints in Planning

Reasoning with constraints is ubiquitous in real-world problem-solving. It is essential in virtually all planning problems.

The central idea behind planning is reasoning about time, action and objects, and constraints over them, for the purposes of influencing the future. Its exclusive definition according to Georgeff [71], Nilsson [120], Sacerdoti [135] and others is the formulation of actions and their aggregation into sequences which when executed by agents (robots) would lead to desired goals. Research in the planning area has concentrated on planning with this view. The emphasis has been on synthesizing sequences of actions for a robot
Theoretical Steps:
1. Identify the problem.
2. Design a method for solving it.

Engineering Steps:
3. Implement the method in a computer program.
4. Demonstrate the power of the program (and thus of the method).

Analytical Steps:
5. Analyse data collected in demonstrations.
6. Generalise the results of the analysis.

Table 1.1: Steps involved in experimental research in AI to execute in interacting with the real world.

There are several planning problems which do not strictly conform to the above definition. They do not require search for fine-grained action sequences (as required by robots) but rather assignments of objects to variables such that some given constraints are satisfied. Take for example, "city planning", where some objects (such as car parks, shopping malls, post offices) are assigned in some space (here, physical space). Each assigned object may impose constraints on other objects around it or even demand the existence of other objects. Many operations research problems (such as scheduling, resource planning, knapsack packing) exhibit these characteristics. They can be numerically modeled and have mathematical solutions. There are other problems such as "travel planning" (as acknowledged by Hayes [80]), "design" [44] and "degree planning" which are atypical planning problems, ill-structured, and cannot be adequately modeled mathematically. In these problems, constraints may be dynamically added to the planning process. Constraints typically specify goals to be achieved by a plan and planning methods make use of constraint satisfaction techniques to achieve the goals. Constraint satisfaction characteristically requires search and in planning, it involves search for a plan from a space of possibilities, which is consistent with a given set of constraints. The constraints guide and limit the search in an "informed" manner.

Little research has been done to date in addressing this class of planning problems. Lack of a general computational model for this class of problems has motivated the research for this thesis.
Chapter 1. Introduction

1.3 Current Research

This thesis investigates the planning of programmes of study for academic degrees as an instance of a special class of real-world "planning" problems. Degree planning is a novel application of AI. It requires satisfaction of a variety of constraints and manifests other technically interesting characteristics which make it a rich domain for AI research. Inadequacy of current AI techniques in simulating a human planner for the generic class of problems to which degree planning belongs motivates development of a novel computational model. Experiments conducted in design and implementation of an intelligent knowledge-based system based on a novel computational architecture are reported. The resulting computational approach is generalised and theoretically abstracted culminating in a novel computational model, called CDSP (for Constraint-Directed Search for Planning), for the class of problems.

Conforming to Buchanan's model of AI research, the class of problems under investigation is identified and characterised. The computational model is then presented prior to its illustration through the motivating example, the degree planning problem.

1.4 Overview of the Research Problem

1.4.1 Degree Planning

In this section, we introduce the degree planning problem [140] which typifies the CDSP class of problems and has motivated the research on CDSP problem-solving.

Degree planning is the process of deriving a programme of courses from the courses offered by a university to be taken by a student in future sessions. The derived programme has to conform to the following specifications.

- It has to be consistent with legislative requirements as stipulated in the university regulations. Such requirements are the formal constraints Form-Cons and are required (or hard) constraints.

- It has to maximally support the individual preferences of a student. The student preferences form the user-added preferential constraints Pref-Cons. They may be hard or soft constraints. Programmes may be revised through the Pref-Cons.

- It must be schedulable i.e., the courses in the programme must fit a temporal sequence for some stipulated time-frame and/or conforming to the student temporal wishes. If a course timetable is given then the derived programme is not to conflict with the timetable.
The objective of a degree planning system is to generate a programme of study $\rho_f$ from some initial partially executed programme $\rho_i$ ($\rho_i$ may be empty) such that $\rho_f$ satisfies all the constraints issued to the planner. A typical degree planning system is summed up schematically in Figure 1.1. Usually, a full-time undergraduate programme requires three years to complete. Each year has two sessions. There are formal restrictions (such as work-load and prerequisite restrictions) on the courses a student can take in each session.

The output from the planning system is a well-formed plan. It is a structured object satisfying all the constraints.

Degree planning is characteristically under-constrained (i.e., it has a combinatorial solution space) with respect to Form-Cons. Like any other planning problem, it can be viewed as a straightforward search problem. It requires search for a plan which satisfies a given set of goals specified by the constraints from a discrete finite space of all possible alternatives. To generate (or guess) and test each possibility is a computationally intractable task i.e., the problem is $NP$-hard\(^1\).

Typically, students have some preferences, heuristics and other knowledge sources such as plans successfully completed by previous students, which they use to systematically construct their plans. Sometimes students take advice from an academic counselor. All these knowledge sources act as constraint repositories from which constraints are opportunistically chosen and can be exploited to reduce search.

In planning, students incrementally fill the plan template (something like $\rho_f$ of

---

\(^1\)Complexities of combinatorial problems are discussed in [88].
Figure 1.1) with preferences until a well-formed plan results. The added constraints may trigger constraint propagation resulting in further constraints being added to or removed from the plan—the addition may be forced by virtue of the prerequisite structure and retraction of some constraints may be effected by violation of other constraints. Violations are usually avoided when the user preferences are composed. If they do occur after the constraints are added, the user resolves them by retracting the culprit constraints and adding new constraints. Possibilities are explored through user addition/retraction of user constraints. Likewise, plan revisions are user-driven through user constraints. In this constraint-directed planning, constraints may become inconsistent. Detection and resolution of the arising inconsistent situations are important concerns encountered by a planner.

Reasoning with constraints is the "heart" of the degree planning problem as search for a plan is guided by a body of constraints. The constraints specify the goals which are to be achieved in the plan and they drive the system. Dynamic addition of user constraints and their exploitation in planning and re-planning are important characteristics of the problem [144]. In addition to a user being the oracle in guiding search, experiential knowledge (such as previous student programmes of study), if available, is also opportunistically used in search reduction.

1.4.2 Other CDSP Problems

There are many practical problems which typify the CDSP class of problems. They include travel planning, scheduling, design problems in architecture, engineering and operations research, resource/task assignment, and time-tabling. User preferences play an important role in solving such problems.

In travel planning, for example, travelers search for a plan which meets most of their preferences. The plans have to satisfy the required constraints such as flight schedules, ticket costs, travel durations, and connection times. The traveler preferential constraints are added until the solution space is reduced to a few acceptable plans. The dynamic addition of constraints requires consistency checking on constraints. If a set of constraints becomes inconsistent, the culprit constraints are relaxed. The dynamic addition of preferential constraints, integration of these constraints in search, decision on which constraint to apply at some given point in search, and constraint-driven exploration of alternative plans and revision of plans characterise travel planning as a CDSP problem.

Humans are able to solve such problems easily because these problems are typically
under-constrained. Search for only a single solution (or few solutions) from a wide space of possibilities is required. Humans use intuitions and heuristics, and also draw upon experiential knowledge to decide on which paths of the search space are promising and worth pursuing.

1.5 Primary Contributions

Traditional planning and constraint satisfaction techniques, and other known standard problem-solving techniques are inadequate\textsuperscript{2} in addressing the degree planning problem, and in general, the CDSP class of problems as characterised above. In the CDSP theory, we provide a model for such problem solving. The proposed theory combines relevant theories from existing techniques and enhances the resulting theory to include support for the additional problem solving dimensions as required by the CDSP problems.

The CDSP theory models the problem-solving approach as required by an intelligent knowledge-based system in simulating a human planner. It is built on a computational architecture in terms of a general problem solving language, called CL-IPS, modeled for the CDSP class of problems. CL is the Constraint Language component of the problem solving language and IPS is the Intelligent Problem Solver component. CL-IPS is implemented in LISP.

All the constraints (including the preferential constraints) and other domain knowledge are formulated as declarative statements in CL and input to the system. These statements are interpreted, internally represented and worked on by a collection of computational mechanisms from the IPS component to compute one or more solutions for the problem.

The user plays an important and essential role in interactively communicating his/her preferences to the system dynamically and in so doing, guiding the problem solving process.

The primary contributions of the research reported in this thesis are summarised below.

- A class of real-world problems (called CDSP) which require planning through constraint satisfaction and for which adequate problem solving techniques are lacking have been identified and characterised.

- A novel problem solving theory (called the CDSP model), which combines and

\textsuperscript{2}The inadequacies of existing theories in addressing the CDSP class of problems are explained in Chapter 3.
Chapter 1. Introduction

extends appropriate existing problem solving theories, is provided for the CDSP class of problems. The theory is provided in terms of a general problem solving language and a novel architecture.

• A declarative problem input language for problem specification and its input to the system.

• An object-oriented knowledge representation language to adequately capture knowledge in the computational model. The model makes use of this existing technology in knowledge representation.

• The CDSP model embodies dynamic constraint satisfaction and in particular, supports reasoning with a variety of constraints which
  - do not have a priori known variables and domains,
  - have a partial ordering between them,
  - are defeasible,
  - are coupled (i.e., constraints which share objects),
  - express both extensional and intensional relationships,
  - are dynamically added and removed at run-time,
  - actively support user-guided navigation and exploration of search space (through user supplied preferential constraints),
  - are exploited in search guidance and search reduction, and
  - may become inconsistent during problem solving.

• Detection and resolution (through relaxation) of constraint inconsistencies.

• Learning and exploitation of experiential knowledge in explanation and advice derivations, and directing problem solving.

• An automatic and intelligent model-theoretic approach to planning and replanning. The search is constraint-directed.

• A systematic, interactive and incremental approach to planning and replanning. This approach ameliorates the model-theoretic approach in that it allows the user to intervene and direct the problem solving. It attempts to simulate human style of problem solving.

• Explanation facilities to provide transparency to the user.
1.5. Primary Contributions

- Novel application of a truth maintenance system in constraint-based reasoning. An ATMS is combined with dependency-directed backtracking to adequately address the problem solving concerns in the CDSP type problems.

- An expert system shell developed for the CDSP class of problems on the CDSP model.

The CDSP problems characteristically delineate combinatorial solution spaces, and hence require reasoning with a variety of constraints to make the search for solutions "informed".

The proposed model supports integration of constraints in limiting the search for solutions. It involves reasoning with constraint hierarchies and constraint associated operators, consistency checking, constraint-directed search for planning, context-sensitive selection of constraints, constraint-directed model search, user-directed navigation of the search space, incremental planning, replanning, advice generation from some existing plan states, and conditional (including counterfactual) query processing. Constraints are also learned from previous cases, if any, and exploited in advice generation and search reduction. Effective integration of constraints in search is an important and essential feature of the computational model.

The model also provides a novel application of a truth maintenance system to manage the constraints. The truth maintenance facility manages the problem solving data, supports defeasible reasoning and constraint relaxation, and facilitates search, incremental planning, conditional reasoning, plan revision and generation of explanations and advice. In addition, it provides semantics for traditional inconsistency detection and resolution. These functionalities are lacking in classical constraint satisfaction formalisms.

All knowledge in the model, including constraints, is represented using a hybrid of frames, logic and rules in an object-oriented representation formalism.

In this thesis, the CDSP model is presented in terms of the general problem solving language CL-IPS. It supports input of problems and interactive acquisition of constraints through CL, employs a state-space representation of the problems, and employs a systematic constraint-directed reasoning mechanism to search for solutions.

The degree planning problem, which is the motivating example of the model and is an instance of the CDSP class of problems, is used to illustrate the model.

In the development and presentation of the model, we address the issue of the construction of automated knowledge-based environments for solving the problems in CDSP viz., knowledge engineering, problem-solving and user interface concerns. The
model conforms to the advisory characteristic [127] in which one (or a few) possible solutions are put forward for user acceptance. If a solution is accepted then the problem is considered solved. Otherwise, another solution is searched for. The user is the final authority on the acceptability of any system proposed solution.

The chief objective of the research is to develop methods for reasoning with a variety of constraints for planning which are not adequately supported by classical constraint satisfaction and planning paradigms. User-directed search for plans and user navigation of solution space (through user constraints) are important and unique characteristics of the problems which are supported in the proposed model.

1.6 Chapter Summaries

This section summarises the contents of each of the remaining chapters.

Chapter 2 reviews previous work on planning and constraint satisfaction relevant to the present research. The present research is placed in the context of current AI research. Existing problem solving models, knowledge representation formalisms, and systems developed for solving relevant practical problems are reviewed.

In Chapter 3, the CDSP class of problems is identified, formally abstracted, and characterised. Inadequacies of existing constraint satisfaction models in addressing this class of problems are discussed. A problem solving language (called CL-IPS) and an architecture for solving the class of problems are presented. This forms the heart of the proposed CDSP model. CL-IPS has two major components: a constraint language CL to state a CDSP problem, and an "intelligent" problem solver IPS which interprets the problem specified in the input language, and organises and performs search for solutions. The CDSP problem solving architecture is presented and its components are described.

In Chapter 4, the encoding and internal representation of CDSP knowledge as objects in a hybrid representation language (called ORL) is briefly reported. Knowledge structures which pertain to CDSP knowledge and which facilitate CDSP problem solving are identified. Also, the methods for deriving these structures are presented and discussed. Syntax and semantics of the general input constraint language CL is presented in Chapter 5. The semantics of the language is given in terms of how statements in the language are used in the CDSP problem solving approaches. Their use depends on their discriminated functional categories.

The IPS problem solving machinery, embodying the solving of Model-Search and IPLAN problems, is presented in Chapters 6 and 7. These problems are two formula-
tions of CDSP problems that make up the CDSP theory.

In Chapter 6, the existing search techniques of immediate relevance to CDSP problem solving are discussed. The properties of constraints useful in guiding search are identified. Upon these properties, the algorithm for automatic selection of constraints is based. The plan evaluation, and plan verification and repair concerns in CDSP problem solving are also addressed. Finally, in this chapter, the Model-Search formulation is characterised and an algorithm is given for it followed by an analysis of the approach.

Chapter 7 presents the formulation and solving of IPLAN problems. In this problem, constraints are dynamically received from a user and exploited in incrementally guiding search for a solution. It uses a single-context assumption-based truth maintenance system and the dependency-directed control regime to facilitate user exploration of solution space, and user-guided incremental search. On exhaustion of all user constraints, IPLAN invokes Model-Search to take some existing partial solution to completion. An algorithm for IPLAN is given with an analysis of its suitability for use in CDSP problem solving.

In Chapter 8, an instance of the degree planning problem is used as a case study to exemplify the CDSP model. The previous attempts at modeling the degree planning problem are briefly reviewed. The Australian National University BSc degree planning domain is formulated as a CDSP. Different kinds of knowledge, including constraints, that pertain to the domain are discussed. Their formalisation in CL and representation in ORL are illustrated. The Model-Search and IPLAN approaches are illustrated using the above degree planning problem.

Finally, in Chapter 9, the results of the research undertaken in developing the CDSP computational model for the special class of real-world constraint-directed planning problems is summarised. Limitations in the current model are acknowledged and possible extensions to overcome them are proposed. Further work to enhance the model resulting in a shell for solving a wide class of CDSP problems is proposed.
Chapter 2

Previous Work

2.1 Introduction

Planning and constraint satisfaction are two distinct areas of problem solving research in AI which are of immediate relevance to the theory of CDSP problems. They have been active areas of research in AI. Both draw on search as the key mechanism for finding solutions.

In this chapter, the planning and constraint satisfaction literature is reviewed with particular emphases on progress made in solving practical problems. The existing constraint satisfaction and planning paradigms are reviewed, and their strengths and weaknesses in solving real-world problems assessed. The chapter begins with a brief discussion of search.

2.2 Search in Problem Solving

Search is a universal problem-solving paradigm in AI. Rich [131] says of AI problem solving that "...many of the problems which fall within the purview of AI are too complex to be solvable by direct techniques; rather they must be attacked by appropriate search methods armed with whatever direct techniques are available to guide the search". Rich gives a comprehensive account of search-based problem-solving methods.

A search process is the sequence of steps required for exploring a space of solutions for a problem and finding a solution for it. A problem space is the environment in which the search takes place [115]. It consists of a set of problem states and a set of operators which transform one state into another. A problem state is a database describing some task-domain situation and an operator is a method or rule of inference to manipulate the database. The problem-solving task involves finding a sequence of operators that map some existing problem state (initial state) to a goal state [8]. Multiple task-domain
2.3. Planning

Nilsson [4] aptly gives the background to planning as follows

Early in the growth of the field of artificial intelligence it was recognized that an important behavior for any intelligent agent was the ability to plan a set of actions to accomplish its goals. The attempt to realize programs with this ability has resulted in one of AI's main subdisciplines—the field of planning.

Simply put, the central idea behind planning is deciding on a course of action prior to acting. The objective of automated planners is to string together a sequence of actions which when passed on to an agent would execute it and produce a desired goal.

Planning is essentially a search problem. The general problem is computationally intractable [71]. Planning defines a potentially large search space and then seeks a point in the search space which satisfies the goals. There are two common ways of
viewing planning as search. One is the state-space search. Here, points in the search space are defined as states of the application's world at various times. Heuristic search is performed on these states and the resulting traversal path from some known initial state to a desired goal state through the application of heuristics (operators) gives the plan. In the second view, planners define points in the search space as partially elaborated plans.

Problems of organizing the search for search reduction, recovery from failure states and drawing on domain knowledge for making informed choices have been the interesting concerns of research in planning ever since the pioneering work of Newell and Simon realised in LT (Logic Theory Machine developed in 1956) [114] and GPS (General Problem Solver developed in 1963) [117] systems.

2.3.1 Planning Techniques

Much of the research effort in planning has focused on representation of planning knowledge and finding ways to reduce search space. *Domain-dependent* planners employ *ad hoc* knowledge representation techniques and exploit domain-specific heuristics for search reduction. *Domain-independent* planners form the bulk of research. They draw on a repertoire of knowledge representation techniques and algorithms which are abstracted from a large variety of real-world application domains.

Several search-based techniques have evolved over the last two decades. They have mostly evolved from previous work. A review of these techniques and a chronicle of planning systems utilising them are given in [160]. In this section, we summarise the important, domain-independent planning techniques and the planning systems in which they are realized.

LT introduced the concept of state-based heuristic search. It worked backwards from the goal, splitting the problem into sub-problems, which it proved separately. GPS generalised this approach into "means-ends analysis" in which operators are opportunistically chosen for application such that the resulting state is closest to the goal state. By using a difference table, the search space is reduced by focusing on only the most relevant operations in achieving the goal state. Many later planning systems (such as STRIPS [55]) have inherited the means-ends analysis technique from GPS.

STRIPS originated from robot planning research and was the first planner to illustrate that much of planning involves search through situations, where a situation is defined as a state of the world usually represented by a list of formulae. It is a member of the general class of problem solvers that search such a space of "world models" to
find one in which some given goal is achieved. In STRIPS the operators are formalised in predicate calculus and represented as rules with pre-conditions and post-conditions. Each problem for STRIPS is a goal to be achieved by a robot operating in some simple world. The solution is a sequence of operators for achieving a goal. It combines the means-ends analysis and formal theorem-proving methods to get more powerful search heuristics than those present in each of these individual methods.

Sacerdoti made a seminal contribution with hierarchical planning. This was realized in his ABSTRIPS [133] and NOAH [134] systems. These systems employed heuristic search by recognizing the most significant features of a problem, developing an outline of a solution in terms of these features, and dealing with the less important details of the problem only after the outline has proved adequate. The solution is formed at the most abstract level and the lower levels are then planned using the upper level abstractions. These systems did not support re-planning or backtracking in the event of failure.

NONLIN [159] is an improved hierarchical planner which supports re-planning or re-consideration of alternatives at any level in the hierarchical space if a solution cannot be found, or if partial solutions indicate that a higher level choice was faulty.

MOLGEN [155] embodies further improvement in the hierarchical planning approach. It is able to opportunistically determine when in the hierarchy a particular choice is sufficiently constrained to be a preferable goal to work on at any time.

SIPE [167] and its successor SIPE-2 [168] have been built on the hierarchical planning methods. They are improvements on previous formalisms in that they support both automatic and interactive generation of plans, extensive use of constraints and resources, efficient methods for representing properties of objects, specification of plan rationale, and ability to express deductive rules for deducing the effects of actions. These systems are interactive—a major improvement on previous planners. Interactive planning is invoked through a constraint language. Constraints are used to represent a wider range of domains.

Opportunistic planning is another influential planning technique. It has its origins in the Hearsay-II speech understanding system [50], and in the OPM journey planning system [82]. It embodies multiple levels of representation, data and goal-directed (bidirectional) problem solving (called opportunistic reasoning—a phrase coined by Hayes-Roth and Hayes-Roth in [82]), and island driving. Opportunistic reasoning supports the ability to start problem solving at any point in the search space, and involves focusing of the planning effort in areas that are of high certainty and/or highly con-
strained. These “islands” are exploited in guiding choice of next best courses of action to pursue. This approach often uses a blackboard architecture through which various components (knowledge sources) communicate via constraint information. The Hearsay-II and OPM systems are built using this architecture.

There are several backtracking approaches available. The efficacy of these approaches depends on the narrowness of solution spaces. In depth-first backtracking (as inherent in PROLOG), states in the solution space at which there are alternative choices are recorded. One alternative is chosen and the search continues. If there is any failure, the saved state at the last choice point is restored and the next alternative taken.

The full potential of truth maintenance systems (TMSs) has not been realized in planning systems to date and currently they form one of the active research frontiers in planning. Doyle’s TMS [49] is a single-environment dependency-directed search approach. It brings about a major improvement in efficiency when compared with the backtracking search techniques. In this approach, states are not recorded and backtracked. Instead, dependencies between decisions, the assumptions which lead to the decisions and alternatives from which a selection can be made are explicitly recorded. Failures are undone by propagating and undoing all of the dependent parts of the solution through the dependency links. Relatively few planning systems have explored and used this approach. Some examples are MOLGEN [155] and O-PLAN [26], and work reported in [21,153].

An assumption-based truth maintenance system (ATMS) [41] succinctly caches out and maintains consistent possibilities (multiple environments) and avoids search. It is a rich “house-keeping” facility and is an important part of the model developed and presented in this research. ATMSs have rarely been used in planning systems. An example on research involving ATMS planning is an extension to ANGUS [162].

The opportunistic search and TMS approaches seem to be potential big contributors to the future of research and development work in planning systems; they are gaining in popularity.

2.3.2 Applications

Temporal and conceptual complexities, and other technical barriers\(^{1}\) in real-world planning problems have restricted planning work mostly to the research laboratories. Plan-\(^{1}\)Fiksel and Hayes-Roth summarise the technical difficulties inherent in constructing automated practical planners.
2.4. Constraint Satisfaction

<table>
<thead>
<tr>
<th>Domain</th>
<th>Planner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Robot Control</td>
<td>STRIPS [55]</td>
</tr>
<tr>
<td>Simple Program Generation</td>
<td>HACKER [157]</td>
</tr>
<tr>
<td>Mechanical Engineers</td>
<td>NOAH [134]</td>
</tr>
<tr>
<td>Apprentice Supervision</td>
<td></td>
</tr>
<tr>
<td>Experiment Planning in</td>
<td></td>
</tr>
<tr>
<td>Molecular Genetics</td>
<td>MOLGEN [68, 155]</td>
</tr>
<tr>
<td>Job-Shop Scheduling</td>
<td>ISIS [60], ISIS-II [61]</td>
</tr>
<tr>
<td>Travel Planning</td>
<td>OPM [82], Globe-Trotter [15]</td>
</tr>
<tr>
<td>Naval Logistics</td>
<td>NONLIN [161]</td>
</tr>
<tr>
<td>Organisational Management</td>
<td>LIBRA [56]</td>
</tr>
<tr>
<td>Spatial Planning</td>
<td>SPECS [154], REEFPLAN [150]</td>
</tr>
<tr>
<td></td>
<td>WRIGHT [11]</td>
</tr>
</tbody>
</table>

Table 2.1: Knowledge-Based Planning Applications.

...ning research is finding its way slowly but steadily into automated systems for solving real-world problems. To date there have been only a few systems which have had some success, albeit in their narrow application domains. Some of these are outlined in Table 2.1.

2.4 Constraint Satisfaction

Constraint satisfaction is an umbrella term for a variety of problem solving techniques in AI and other disciplines (such as Operations Research). It is a relatively newly recognised area of AI and hence there has not been a great deal of research or applications undertaken in it from the AI perspective.

In the following, we summarise the relevant issues addressed, research done, techniques developed and real-world applications in the constraint satisfaction area.

2.4.1 Characteristics

Constraint satisfaction involves reasoning with and about constraints. Constraint satisfaction problems (CSPs) are essentially search problems in which given an entire space of objects, whole classes of them are eliminated at a step by applying successive constraints until the set is narrowed down to objects that satisfy all the constraints. New objects are not created but objects are filtered in the process.

Mackworth [96] dichotomises CSPs roughly into Boolean constraint satisfaction problems (BCSPs) and constrained optimisation problems (COPs).
A BCSP is modeled as follows: given is a set $X$ of $n$ variables $\{X_1, X_2, \ldots, X_i, \ldots, X_n\}$, associated with each variable $X_i$ is a domain $D_i$ of possible values for $X_i$. On some specified subsets of these variables there are constraint relations $C(X_i, X_j, \ldots)$ given which are subsets of the Cartesian product $D_i \times D_j \times \ldots$. The set of solutions is the largest subset of the Cartesian product of all the variable domains such that each $n$-tuple in that set satisfies all the given constraints. The BCSP is unsatisfiable if the solution set is empty.

In the BCSP formulation, all the variables and their domains are known prior to the search. The constraining relations are extensionally described. Some example problems are $n$-queens, map-colouring and crossword puzzles.

A COP is a numerical optimisation problem in which the solution sought maximally satisfies a known set of local constraints. It may involve discrete and/or continuous variables and constraint relations may be specified both extensionally and intensionally. Some examples are design, scheduling, scene labeling and cutting-stock problems.

Many problems can be cast into either or both of the above classes. There are some CSPs however which cannot be adequately captured in either formulation. Some of these problems form the impetus for the current research and are discussed in depth in Chapter 3.

Generally, CSPs in AI are typically computationally intractable since Boolean satisfiability is NP-hard [69]. A general algorithm would require exponential time. Recent research in constraint satisfaction has mostly concentrated on BCSPs with the view of developing domain-independent problem solving approaches which use constraints to avoid combinatorial explosion. Several search-based and constraint propagation approaches have emerged.

### 2.4.2 Search-Based Approaches

Hentenryck [165] classes the search-based approaches as a posteriori or a priori.

In a posteriori approaches, constraints are used passively. Generate-and-test and backtracking [72] form the kernel for most of such approaches.

In generate-and-test, each possible combination in the finite assignment space $D = D_1 \times D_2 \times \ldots D_i \ldots \times D_n$ is systematically generated and then tested to see if it satisfies all the constraints. The first combination which satisfies all the constraints is

---

2Extensional descriptions refer to descriptions of domains in which the number of objects is finite and the objects are explicitly defined.

3In intensional relations, objects in the domains are not explicit and are described qualitatively.
the solution. No pruning occurs in the search. Thus generate-and-test explores the search space exhaustively, and although the algorithm is correct, it is very inefficient. DENDRAL [94], constructed for structure elucidation of organic compounds, embodies the generate-and-test algorithm in one phase of its problem solving.

Standard backtracking is an “uninformed” tree-search approach embodying depth-first search with chronological backtracking. Given a sequence \( < x_1, \ldots, x_k > \), the problem is to extend it by finding a value \( x_{k+1} \) for \( X_{k+1} \) such that the constraints are not violated. If there is not such a value then no assignment to the remaining variables can possibly resatisfy the violated constraint. In other words, a failure (or dead-end) situation is reached in search. In such cases, the last variable that was assigned is reassigned to the next legal value and consequently, \( a posteriori \) pruning results. While the efficiency of backtracking search is much better than that of generate-and-test, it can still be very poor [84].

Inherent pathological behavior (known as thrashing\(^4\)) in standard backtracking have focused research on devising schemes for improving these approaches by making them “informed”. Consequently, numerous schemes have been developed \( viz, \) lookahead schemes (involving variable ordering and value ordering [40, 66, 77, 126]), and look-back schemes (involving transfer of control back to source of failure and searching, and value recording) [90].

Dependency-directed backtracking [153] and truth maintenance systems [29, 49, 102] realise these improvements, allowing choices to be undone independent of the order in which they were made. de Kleer [32] compares the interrelationships between the ATMS and CSP techniques and argues that both the approaches embody similar intuitions in attempts to avoid the thrashing malady in backtrack search.

A major improvement in efficiency (proven theoretically and experimentally [165]) has been brought about by a set of consistency-based approaches [37, 65, 95]. These approaches attempt to improve search efficiency by pruning the object domains and constraint relations \( a priori \), leading to a whole family of hybrid tree search/consistency algorithms [110, 111]. In these approaches, preprocessing is done to minimise backtracking and constraint checks by early detection of failures. In the preprocessing phase, node, arc and path inconsistencies [95] are detected and resolved prior to running the search. This prunes the search space by removing combinations of values that cannot appear in a solution.

Ideas on consistency-based approach originated from Waltz’s filtering algorithm.

\(^4\)Thrashing and its symptoms are explained in [96, 165].
Chapter 2. Previous Work

[166] and Fikes’s REF-ARF [54]. They were explored and further developed in [65, 70, 77, 107], and also used in ALICE [92]. Barzel and Barr [9], Bessière [12, 13], and Dechter and Dechter [39] have studied the consistency techniques in *dynamic constraint satisfaction problems*.5

### 2.4.3 Constraint Propagation Approaches

Constraint propagation is a deductive activity in which results from satisfaction of a constraint $C_i$ from a sequence of constraints $< C_1, C_2, \ldots, C_i, \ldots, C_n >$ are forwarded to satisfaction of $C_{i+1}$. This activity is either embodied in problem solvers or in specialist subsystems (such as truth maintenance systems) coupled to problem solvers. Constraint propagation is characterised by *label inference* [27] and *value inference* [28, 158].

In label inference, constraints are used to reduce the set of possible variable assignments by restricting the domains for the variables. Mackworth’s preprocessing consistency techniques (as discussed above) fall into this category. This technique has been used in several applications. Wilkin’s work on planning [167] and Waltz’s work on vision [166] are two examples of such applications.

In value inference, the values already assigned to variables are used to deduce values for unassigned variables. It characterises a data-driven approach to inferencing and use of constraints as soon as they are applicable. Use of the constraints by propagating values in this manner reduces the search space *a priori*. SKETCHPAD [158], THINGLAB [14] and CONSTRAINTS [156] use this approach.

An ATMS [30] also embodies constraint propagation during search. The present research requires the services of an ATMS. Its characteristics and use in the present research are explained in Chapter 6 of this thesis.

### 2.4.4 Constraint-Satisfaction Systems and Applications

Research in constraint satisfaction has produced several successful problem solving theories. Although much work has been problem-specific, research work has been actively pursued in developing domain-independent theories. Constraint programming languages are also emerging from various research laboratories. In this section, some of the constraint-satisfaction systems and constraint languages are reviewed, and some successful applications are referenced.

---

5The CDSP problems are dynamic CSPs. The relevant literature is analysed in Chapter 4 where the CDSP models are developed.
2.4. **Constraint Satisfaction**

2.4.4.1 **Problem-Solvers for CSPs**

REF-ARF [54] and ALICE [92] are influential problem-solvers developed for solving combinatorial problems requiring constraint satisfaction. Each of them have a particular language in which the user states a problem. The systems interpret the statements in the respective languages and use the idea of *a priori* pruning to find a solution. Although these systems have been successful in their application domains, they have several drawbacks in their application to other general problems. Hentenryck critiques these systems in [165].

In REF-ARF, a nondeterministic programming language (REF) is used to state a problem and a problem-solver (ARF) interprets and solves the problem stated in REF. REF-ARF's task involves a heuristic search for values for a set of variables that satisfies all the constraints given by linear equations. The generate and test approach is used in which the constraints are exploited in the generation process for reducing the search space.

In ALICE, problems are stated in a mathematical language based on set theory, first-order logic, and some notions of graph theory. The user asks the system to compute a function from one finite set to another, which satisfies a set of constraints. The system draws on a given repertoire of general heuristics and uses a depth-first search with sophisticated constraint manipulation techniques to reach a solution.

2.4.4.2 **Constraint Languages**

In an attempt to solve classes of constraint satisfaction problems from a different perspective to writing specialised programs in procedural languages, there has been a proliferation of general-purpose, high-level languages embedding the constraint satisfaction techniques in recent times. There is an active constraint logic programming community pursuing research in this area [16,23,25,85,136,147]. Its objective is to extend the logic programming languages to include constraint processing. Most of the languages developed support incremental constraint satisfaction; they support dynamic addition and deletion of constraints at run-time.

Jaffer and Lassez [85] describe a scheme CLP($D$) for Constraint Logic Programming Languages (CLP), which is parameterised by $D$, the domain of constraints. In this scheme, the constraints are accumulated and tested for satisfiability over $D$, using the techniques appropriate to the domain.

CHIP is a CLP language [46,84] which uses consistency and case analysis algorithms [95] for reasoning with constraints. It solves, flexibly and efficiently, a large class
of combinatorial problems. It has been successfully applied to graph colouring, car-
sequencing, warehouse location, cutting-stock, and digital circuit diagnosis problems
(*ibid*).

CHARME [136] is a CLP which has been developed to solve a family of scheduling
and resource allocation problems. CHARME has been built from CHIP and latest
research work on constraint processing. The language is declarative and allows the
user to state the problem as a set of constraints on specific objects. Once the problem
has been stated, CHARME provides the user with a set of dedicated primitives to search
for the solution. CHARME is a rich language and allows for further user definition of
procedures and constraints within its language framework. It attempts at providing a
rich environment for solving a family of AI and Operations Research problems.

The consistency techniques have also found their way into a commercial object-
oriented constraint-based programming tool called PECOS\(^6\). PECOS has been devel-
oped to efficiently solve problems in AI and Operations Research. These problems
include resource allocation, planning, scheduling and placement optimisation. It pro-
vides support for dynamic constraint posting, object-oriented programming, search
primitives, optimisation procedure based on branch and bound, non-deterministic pro-
gramming, backtrackable data structures, and integration with LISP.

Galileo [16] realizes a novel approach to logic programming in which constraints are
stated as sentences in a language called *first-order free logic* (an extension of first-order
logic) and their satisfaction involves finding a model for the sentences.

Echidna [146, 147] is a new CLP language which uses consistency algorithms to
process a wide variety of numeric constraints. A unique feature of Echidna is that
it implements domains for real-valued variables with hierarchical data structures and
exploits this structure using a hierarchical arc consistency algorithm specialised for
numeric constraints.

Other constraint languages include CONSAT [76], Bertrand [93], CONSTRAINTS
[156] and THINGLAB [14].

2.4.4.3 Applications

Constraint-based systems have been applied in several areas. Some of these areas in-
clude computational vision [97, 166], job shop scheduling [62], telescope scheduling [86],
experimental planning [155], academic scheduling [125], spatial planning [10, 11], geo-
metric manipulation [24], design [17, 63, 75], and graphics [14, 158]. Further constraint

\(^6\)PECOS has been developed by ILOG in France on work reported in [42].
satisfaction systems are given in [46, 74, 76, 78, 79, 156].

Stefik's MOLGEN and Fox's ISIS are of immediate relevance to the present research. In all these systems constraints are exploited in problem solving.

MOLGEN is a successful application of the constraint propagation approach. It plans gene-cloning experiments in molecular genetics using a hierarchical planning approach (called constraint posting) which uses constraints to represent interactions between subproblems. Constraints are used as elimination rules for eliminating solutions, as commitments made by the planner to partially describe solutions, and as a communication medium for expressing interactions between subproblems. Variable value constraints from refinement subproblems is propagated to other subproblems. Constraint propagation makes possible a least-commitment of deferring decisions as long as possible. The accumulated constraints and variables are used in the constraint satisfaction which plays a coordinating role by pooling the constraints and intersecting their solutions.

ISIS uses constraint-directed search for job shop scheduling. It takes relevant scheduling information as constraints on a schedule and attempts to satisfy all these constraints. If a schedule which satisfies all the constraints cannot be found then the unsatisfiable constraints are identified by the system and alternatives are generated. These alternatives represent relaxation of the constraints.

A frame-based system to model a job shop production environment at all levels of detail is used by ISIS. The job shop scheduling problem delineates a wide search space of schedule possibilities. The a priori constraints are used to narrow the search space. The search for a schedule is performed hierarchically, by trying to satisfy all the constraints, beginning with the most important ones. Also, it has the capability to reason forward or backward from any point in the search space. This bidirectional scheduling is aptly known as opportunistic scheduling.

2.5 Current Work in Context of Previous Research

The primary concern in the present research is the development of a computational model (CDSP model) for a specialised but useful class of practical planning problems which require constraint satisfaction. The computational model is abstracted so that it is applicable to a wide range of problems. The major components of the model are a general constraint language, a knowledge representation formalism for the representation of constraints and other planning knowledge, and algorithms and knowledge structures for reasoning and capturing the reasoning process respectively.
Chapter 2. Previous Work

The computational model proposed is a novel attempt to combine some of the existing planning and constraint satisfaction theories as necessitated by the class of problems under investigation. It borrows the notions of frame-based knowledge representation, integration of constraints in directing search, constraint propagation, and opportunistic problem solving. Novel features, including extensions to existing theories, in the model include a rich input constraint language for formulating problems in the CDSP class of problems, reasoning with intensional and extensional constraints, use of experiential knowledge as constraints on the search space, heuristic-based selection of constraints from a set of active constraints for application at different points in the search space, use of an ATMS to manage constraints and problem solving data and maintain consistency, use of a browser to make points in the search transparent to the user, incremental planning, plan revisions, and conditional query processing. The constraint language captures all the constraints (both fixed and user preferred) and provides the medium for the user to explore the search space.

Although there are many classes of real-world problems which require satisfaction of constraints, much of the constraint research done to date has been application-specific. There does not exist any general theory for constraint representation, nor use of constraints in reasoning including approaches for detection of constraint violations and their resolutions. Also, surprisingly, very few constraint-based systems have been developed which allow users to add their preferences dynamically at run-time and freely explore the space of solution possibilities. Many systems are rigid in that they produce a solution and if it is not acceptable by the user, search for solutions anew. They are not incremental, and do not support dynamic addition of constraints and revisions.

In conclusion, there have been many problem solving theories proposed for constraint satisfaction and planning problems separately. Only a few of these theories have combined to deal with the planning problems which utilise constraints during the course of problem solving. ISIS, MOLGEN and SIPE are examples of the few systems which embody exploitation of constraints in problem solving. ISIS and MOLGEN are domain-specific planners whereas SIPE is an attempt to model constraint-directed practical planning for abstract domains.

The computational model presented in this thesis combines the relevant existing theories and extends the resulting theory to capture the problem solving in the CDSP class of practical planning problems. It is a novel attempt at providing a general problem solving language which supports a declarative formulation of the problems in the CDSP class in an input language, a robust knowledge representation formalism, and
computational mechanisms which draw on knowledge contained in problem constraints for guidance in search for solutions. It embodies a theory of constraint-directed search based on a variety of both quantitative and symbolic constraints. An important practical feature of the model is its support for user-guided navigation and exploration of solution space through the input language. Degree planning is a novel application of the constraint satisfaction and planning technologies and is the motivating problem for the model.
Chapter 3

CDSP Model

3.1 Introduction

This chapter introduces the CDSP (Constraint-Directed Search for Planning) model. We motivate the CDSP model with an illustration of an approach which might be adopted in solving a CDSP type of problem. The model is briefly explained in terms of its constituent knowledge levels. The CDSP class of problems is then characterised and major issues of importance in modeling CDSP problem solving are identified. Inadequacies in existing reasoning models relevant to CDSP problem solving are argued and the need for a novel computational model motivated.

In the final section, a formal statement of the problems is given, the problem structure discussed, and the CDSP computational model proposed for the class of problems in terms of a general problem solving language (called CL-IPS) which comprises of an input language component, CL, and a problem solving component, IPS. A system architecture based on this model is presented and its various components introduced. The CL and IPS components of the model are presented in detail from Chapter 5 onwards.

3.2 Solving CDSP Problems

In this section, we describe a problem solving approach which may be adopted in solving CDSP class of problems. An identification and top-level description of the tasks that are to be performed by an automated problem solver are given, and a generic CDSP problem solving model is characterised. The constraints are characterised in the next section.
3.2. Solving CDSP Problems

3.2.1 A General Approach

In CDSP problem solving, the user creates a plan as an artifact within the context of a set of formal constraints and using his/her preferences. The formal constraints are usually known \textit{a priori} to problem solving. The user develops an acceptable plan incrementally by adding preferential constraints reacting to inconsistencies that may consequently arise until all formal and preferential constraints are satisfied. The user generally formulates new preferential constraints (using knowledge of the domain) which are directed towards satisfying the formal constraints and any already stated preferential constraints. However, the user may overlook \textit{support constraints} which may render the resulting set of constraints inconsistent. These situations are resolved by the user retracting or relaxing the culprit preferential constraints.

In this approach to CDSP problem solving, the user explores possible plans allowing plan revision through the use of preferential constraints. In developing a plan the user draws upon experiential knowledge and other available knowledge sources for heuristics to assist in focusing on promising solutions, and avoiding any previously detected inconsistencies.

An efficient and effective search for a solution is achieved through informed formulation of preferential constraints.

3.2.2 Tasks in CDSP Problem Solving

Given a set of constraints, a set of plan objects, and a set of domain heuristics, the objective of CDSP problem solving is to search, exploiting constraints as heuristics for focusing search, for a plan which is consistent with all the constraints. The search is viewed as a constraint-directed selection of plan objects and a distribution of the selected plan objects into a sequence of bins. There are at least two such bins and they may be contiguous or overlapping. The filling of these bins is the scheduling aspect of the search.

The constraints are differentiated functionally and used in different parts of the search. The constraints specifying inclusion of plan objects are used systematically for the selection process, and the temporal constraints (such as the length of the bin sequence) and the constraints on bins (such as bounds on the number of objects or total weight of objects that can be placed into the bins) are used in the scheduling process.

During the search for a solution, the constraint-directed selections of plan objects and their support structures are derived and captured as a cumulative structure—a
cumulative structure is progressively developed during search. (The various structures used in search are explained in Chapter 4). Consistency is checked on each resulting cumulative structure. It includes checking the schedulability of the cumulative structure by finding a schedule for a cumulative structure if one exists. If a cumulative structure is found to violate any of the constraints then other selections for previously selected structures are explored through an "informed" backtracking.

### 3.2.3 CDSP Model: A Top-Level Description

Figure 3.1 distinguishes the multi-layered structure of the CDSP model. Problem, domain and the generic problem solving levels make up the complete CDSP model.

At the problem level, the problem-specific information is communicated to the problem solver through an input language. *preferential constraints* is a set of user supplied preferential constraints. *xplan* is an executed section of a plan. It is the committed section of a plan and has to appear in all plans searched. *plan* is an existing plan. If *plan* is given and is no longer acceptable then it is minimally revised. Instances of problems belong to the problem level. For example, an instance of the degree planning problem (i.e., planning a degree for one student) or travel planning problem (i.e., planning a
3.3. A Characterisation of CDSP Problems

travel itinerary for one traveller) belong to this level.

At the domain level, the domain-specific information is contained. The *formal constraints* are defined for a domain and they remain fixed during problem solving for all the problems in the domain. Also, the *plan objects* and *knowledge sources* (domain-specific heuristics) belong to the domain level. This information is communicated to the problem solver in the same input language as the problem level information. In the degree planning domain, the legal requirements form the formal constraints and all the courses available for the set of plan objects. In the travel planning domain, flight timetables form the formal constraints and the actual flights form the plan objects.

The problem solving level is the fixed generic CDSP problem solver. It receives constraints and instructions from the other two levels and processes them. The problem solving engine has two components: a set of search operators given by $\Gamma$, and a search organiser given by $\mathcal{K}$. $\mathcal{K}$ is a control operator which embodies domain independent heuristics for the CDSP class of problems. At each point in the search process, it determines the next best constraint to hand over to $\Gamma$ for application. $\Gamma$ is a set of operators for constraint interpretation.

3.3 A Characterisation of CDSP Problems

The CDSP class of problems are real-world, practical planning problems which have goals specified by a set of a variety of constraints, $\Delta$, expressed as sentences in an input language. Solutions to these problems are found by searching for a point in a space of possibilities which satisfies $\Delta$. The problem solving involves the search for a *plan* (or *solution*), $\rho$, such that $\forall c \in \Delta$, the predicate, *satisfies* ($\rho$, $c$), is true (this two place predicate tests whether or not a given plan $\rho$ satisfies some constraint $c$). The notation $\text{Plan-Objs}$ denotes the complete, finite set of objects (through their unique identifiers), that can be included in a plan; and $B_i$ denotes a particular subset of $\text{Plan-Objs}$; $\rho$ is a finite sequence of such subsets (or bins of plan objects as characterised earlier): $< B_1, B_2, \ldots, B_i, \ldots B_n >$.

This amounts to finding an interpretation of the set of constraints, $\Delta$, which satisfies all the constraints in $\Delta$. We define an *interpretation* of $\Delta$ as a mapping from $\Delta$ to a finite sequence of sets of plan objects from $\text{Plan-Objs}$. We introduce the concept of a *consistent* $\Delta$ to mean that there is an interpretation which makes each constraint in $\Delta$ true. It is *inconsistent* otherwise. We call an interpretation which satisfies every constraint in $\Delta$ a *model* of $\Delta$. Hence, a *plan* (or *solution*), $\rho$, is a model of $\Delta$.

In the following subsections, we identify characteristics of CDSP problems. The first
three are CDSP defining characteristics and the remaining characteristics are typical CDSP problem solving characteristics.

### 3.3.1 Plans

In all CDSP problems, the objective is to ultimately find a sequence of bins: \( \rho = < B_1, B_2, \ldots, B_i, \ldots, B_n > \), containing plan objects (as activities) in each of them. The plan has to be consistent with various kinds of constraints all of which will be discussed in latter sections. In summary, the following properties are characteristic of the plans.

- \( \rho \) is a linear (i.e., non-conditional) plan.
- The objects in \( \rho \) are well-supported (i.e., sufficient conditions for the existence of each object in the plan are met) and they have a partial ordering between them.
- The objects in \( \rho \) have a partial ordering between them. The partial ordering specifies that plan objects in the earlier bins in the plan have to be executed prior to objects from the latter bins.

\( \rho \) is consistent with all the constraints defined for the problem. All the non-scheduling constraints are satisfied through the selection process. In the search for schedules, all the precedence, coincidence, bounds on each bin and plan time constraints are satisfied. Constraints and other domain-specific heuristics are employed in making search for a plan efficient.

In degree plans, the plan objects are units which are planned to be undertaken and the bins are the sessions (or semesters). A travel plan forms a degenerate case of the general CDSP plans. In it, the plan objects are particular flights and each bin represents a sector. Each bin contains a single flight.

### 3.3.2 Plan Objects

The set \( \text{Plan-Objs} \) consists of plan objects which are uniquely identified and which may have attributes. Constraints on selection of plan objects are usually expressed as predicates on these attributes.

A characteristic of the CDSP problems is that each plan object is assigned a numeric weight. An aggregate constraint specifies bounds on the sum of weights of a certain set of plan objects. Also, the objects have supports which capture precedence and other temporal relations between plan objects. Support for a plan object, \( O \), consists of a set of plan objects which have to be present in a plan if \( O \) is to be present in the plan. A plan object may have multiple, alternative (i.e., disjunctive) supports.
3.3. A Characterisation of CDSP Problems

In degree planning, particular units of study are the plan objects, and each unit has a weight, and possibly alternative sets of prerequisites. In travel planning, travel sectors are the plan objects, sector fares or durations are weights, and precedence relations on travel segments are the supports.

3.3.3 Constraints

Much of the knowledge in the CDSP problem domains consists of constraints which describe relations that must be maintained in any solutions to the problems. The constraints characterise goals to be achieved in the problem solving. Each of the constraints map to a set of multiple solutions each of which is consistent with the constraint. In real-world CDSP problem solving, information contained in constraints is used by humans to guide the generation and selection of plans. Usually, a variety of quantitative and qualitative constraints are found in these domains. The quantitative constraints are given as algebraic expressions (e.g., “sum of points from first-year is at-least 6”) and the qualitative constraints are symbolic (e.g., “include comp1001”).

We may further characterise constraints as one or both of extensional and intensional constraints. An extensional constraint is a constraint which specifies constraining relations on domains which are defined explicitly as a finite set of discrete values. For example, in degree planning, we may have a constraint “include comp1001” or in travel planning, we may have “take flight FJ911”. An intensional constraint is a constraint which specifies constraining relations on domains implicitly in descriptive terms. For example, in degree planning, we may have a constraint “include units from computer science” or in travel planning, we may have “visit Los Angeles”.

Constraints are also either formal or informal. The formal constraints are required to be satisfied by all plans, and form the hard constraints that cannot be relaxed. The informal (or preferential) constraints are soft constraints that can be relaxed. In this sense they are defeasible. (In this thesis, required and hard, and preferential and soft are interchangeably used to mean formal and informal respectively).

A particular constraint can thus be categorised as either qualitative or quantitative, extensional or intensional, and formal or informal. A set of constraints may also be ordered in a hierarchy based on their specificity.

Further, in CDSP problems, some constraints may be defined locally on each temporal interval in a plan or globally on a plan. In degree planning, “take at most 4 units in year1 session1” is an example of a local constraint. “take at least 12 science points” and “plan time is between 3 years to 6 years” are examples of global constraints.
A plan is *admissible* if it satisfies all the formal constraints. Admissibility determines the legality of a plan against the formal constraints. A plan is *acceptable* if it is admissible and satisfies all the preferential constraints. The user explores the space of admissible plans to identify an acceptable plan.

![Diagram of CDSP model constraints](image)

**Figure 3.2: Summary of Typical Categories of Constraints in CDSP Problems.**

It is useful to particularly identify functional groups of constraints typically found in the CDSP problems (see Figure 3.2). Each is described below.

**Inclusion**

These constraints specify plan objects for inclusion in a plan. They are satisfied if the specified objects are present in a plan. If they are not satisfied, the specified objects have to be added which may force inclusion of other objects via supports. They are formal or informal, qualitative or quantitative, and extensional or intensional.

**Exclusion**

These constraints specify plan objects which are not to be present in a plan. They are satisfied if the specified objects are absent from a plan. They are informal, qualitative, and extensional or intensional.
3.3. A Characterisation of CDSP Problems

Support

These constraints are defined as attributes of plan objects. They demand the existence of certain other objects in a plan. They include two classes of constraints. These are the co-requisite and prerequisite constraints. Their properties are given below in terms of two distinct plan objects, $O_a$ and $O_b$.

1. If $O_b$ is a co-requisite of $O_a$, written $O_a \# O_b$, and if $O_a$ is placed in some bin $B_i$ of $\rho$ then $O_b$ must be placed in $B_l$ where $l \leq i$.

2. If $O_b$ is a co-requisite of $O_a$ then it does not necessarily mean that $O_a$ is a co-requisite of $O_b$. That is, $\#$ is not a commutative relation.

3. The co-requisite relation is defined only on plan objects.

4. If $O_b$ is a prerequisite support to $O_a$, written $O_b \prec O_a$, then, if $O_a$ is included in $B_i$, its prerequisite support, $O_b$ must be included in one of the sets preceding $B_i$.

5. The prerequisite support for a plan object $O_a$ may be another plan object $O_b$ or a constraint $C$. $O_b$ is called a ground prerequisite support for $O_a$. The constraint supports have the semantics that the plan objects chosen for the satisfaction of these constraints have to be placed in the sets containing the constraints or the sets preceding them.

6. An important property of prerequisite supports in CDSP problems is that they are non-circular. That is, given distinct objects $O_a$ and $O_b$, if $O_b$ forms a support for $O_a$ then $O_a$ does not belong to the closure of prerequisite supports for $O_b$. Hence, object supports aptly characterise directed acyclic graph structures conceptually.

7. The support relations are many-to-many relations i.e., there may be multiple supports for some objects and there may be multiple objects supported by some objects; likewise, there may be multiple co-requisites for some plan object and some co-requisites may be co-requisites to many plan objects.

Incompatibility

These constraints specify the objects which cannot exist together in a plan. Presence of certain plan objects in a plan may demand non-inclusion of certain other plan objects in the plan. These constraints are formal, qualitative and extensional.
Temporal

These constraints specify temporal restrictions which have to be satisfied, such as the position of the placement of objects in the discrete temporal intervals with regards to their availability restrictions, any user-specified coincident and precedent constraints; and satisfaction of any global temporal restrictions such as restrictions on plan time. Examples from degree planning for the coincident, precedent and plan time constraints, respectively, are “take unit comp2012 before unit comp2013”, “take unit comp2012 concurrently with unit comp2014” and “plan time is between 3 to 4 years”.

The availability constraints specify that a plan object has to be available in the $i^{th}$ temporal interval if it is to be assigned in $B_i$. A plan object may be available in single temporal intervals, multiple temporal intervals, or over a sequence of multiple temporal intervals.

Coincident and precedent are temporal notions. A coincident constraint specifies that some given set of plan objects have to be assigned in the same given temporal interval. A precedent constraint specifies that a given plan object has to be assigned to a temporal interval that precedes the interval to which some other plan object is assigned. The co-requisite constraints are both coincident and precedent in that if “plan object $b$ is a co-requisite of plan object $a$” then $b$ may be assigned to an interval to which $a$ is assigned or $b$ may be assigned to an interval preceding an interval to which $a$ is assigned. The prerequisite constraints are precedent constraints.

All temporal constraints are formal or informal, qualitative, and extensional or intensional.

Knapsack

The aggregate constraints are essentially knapsack constraints [69]. They specify upper/lower bounds on the aggregate weights of classes of objects (discriminated by object attributes) in a plan. These constraints specify algebraic relations and are formal or informal, quantitative, and extensional or intensional. They belong to both the sets of inclusion and exclusion constraints.

The knapsack constraints are algebraic expressions such as

1. $\sum_{pred_z} x_{wt} \leq M$ for upper bound,
2. $\sum_{pred_z} x_{wt} < M$ for strict upper bound,
3. $\sum_{pred_z} x_{wt} \geq M$ for lower bound,
4. $\sum_{pred_z} x_{wt} > M$ for strict lower bound, and
3.3. A Characterisation of CDSP Problems

5. \( \sum_{pred_x} x \cdot wt = M \) for equality.

In these expressions, \( pred_x \) denotes a set of all those plan objects \( x \) in some given plan which satisfies some predicate \( pred \), \( wt \) is some numeric weighting of the objects, and \( M \) is a real number. Constraints of the kinds given in 1 and 2 are called the upper limit constraints whereas constraints of the kinds given in 3 and 4 are called the lower limit constraints. Constraints of the kind given in 5 are called equality constraints. Both the upper limit and lower limit constraints constitute inequality constraints.

The knapsack constraints also specify upper bounds on the contents of each bin.

Committed

These constraints are derived from an xplan, if available. The plan objects already executed and hence committed act as constraints on search. These constraints specify that all potential plans must include the committed objects unless the user explicitly retracts the plan objects from the plans. They are defeasible and hence informal. They are also qualitative and extensional.

Most of the constraints from the above functional categories may be defined both locally or globally. Only the upper limit knapsack constraints are specifically defined on each of the bins. The availability constraints and the coincident constraints are exclusively local constraints.

3.3.4 Incrementality

In CDSP problems, a plan is usually developed incrementally and systematically. The search for an acceptable plan entails progression towards satisfaction of goals which are specified by constraints. Each search step from some point in the search space results in some addition to the plan information and moves the search closer to the goals. Hence, the plan structure gets elucidated progressively through local search for constraint satisfaction, in much the same way that a treasure hunter "closes in" on a target following a series of local decisions, each based on new information gathered on the way.

For example, the knapsack constraints in the CDSP problems embody restrictions on aggregate weights from selections of groups of plan objects which satisfy some selection criteria based on object attributes. The use of these constraints in searching for an acceptable plan requires selection of those plan objects which increase weights of
appropriate existing group weights until the restrictions prescribed by the constraints are met.

The search is systematic in that it carries the plan forward to full elucidation efficiently. If an inconsistency arises in the problem solving, information contained in an existing plan is usually used to recover from the failure. Rarely, plans are constructed anew.

### 3.3.5 Computational Intractability

CDSP problems are characteristically massively under-constrained with respect to the formal constraints, and hence, possess combinatorial solution spaces. In general, given a $\Delta$, there are usually multiple solution possibilities. Searching for one solution using the generate and test search paradigm is intractable. For example, in BSc degree planning at the ANU, an estimated upper bound of $10^{50}$ plans are possible.

### 3.3.6 Constraint-Directed Search

CDSP planning is characteristically driven by the constraints. Information contained in the constraints is exploited in directing search for acceptable plans. Also, there are usually multiple ways of satisfying each of the constraints in CDSP problems. This gives rise to non-determinism in the search for a solution.

### 3.3.7 Constraint Propagation

Search for plans by applying operators to satisfy certain constraints may result in satisfaction of certain other existing constraints (because some constraints may be coupled) and/or addition of further new constraints to the search process. These consequences get propagated through latter application of appropriate constraint operators in search.

For example, in degree planning, to satisfy “take unit comp3012”, the unit comp3012 has to be added to the plan. comp3012 requires its supports to be added to the plan. In this way, constraints get propagated through addition of new constraints. Also, if the constraints $C_1$: “take science units worth at least 12 points” and $C_2$: “take second year mathematics units worth 4 points” are given, and it is declared that mathematics is a science subject, then selection of plan objects to satisfy $C_2$ will result in taking the plan closer to the satisfaction of $C_1$. In some cases, satisfaction of some constraints may result in satisfaction of others. This explains coupling between constraints.

---

1. A problem is under-constrained if it has multiple solution possibilities.
3.3.8 Dynamic Constraint Satisfaction

Interactive addition of preferential constraints to the search process gives a dynamic character to CDSP problem solving. These constraints are opportunistically integrated in the search process to focus search on promising regions of a search space. The preferential constraints allow user exploration of the solution space.

3.3.9 Exploration of Solution Space

A novel feature intrinsic to CDSP problem solving is the support for the user guided exploration of the solution space. This is achieved through conditional reasoning (including counterfactual reasoning) and is required for conditional query processing. The conditional queries are formulated in terms of preferential constraints and interactively communicated to the problem solver.

3.3.10 Inconsistency

Inconsistencies in $\Delta$ may arise during CDSP problem solving. Their avoidance through a priori checking, early detection, and resolution (i.e., a posteriori recovery) are important computational concerns.

3.3.11 Constraint Relaxation

In CDSP problem solving, an exhaustive search for some $\Delta$ may not result in a model (i.e., $\Delta$ may be inconsistent). In these cases, the culprit constraints from $\Delta$ have to be identified and relaxed under guidance from the user. The constraint relaxation process involves retraction of culprit constraints, or replacing a more constraining culprit constraint with a less constraining one.

3.3.12 Plan Revision

Revision of plans in light of changing reality is an important characteristic of human style of real-world problem solving. In CDSP problem solving, revision of plans become necessary when the user expresses through preferential constraints that some existing plan needs to be minimally changed to become acceptable. It also becomes necessary when some existing set of constraints is found to be inconsistent or when constraints get retracted or deactivated. (Only the preferential constraints can be retracted.)
3.3.13 Plan Verification and Repair

Revision in some existing plan is also prompted if it is inconsistent with an updated set of formal constraints. Existing plans need to be regularly verified and optimally revised in light of any contingencies. For example, in degree planning, formal constraints on degree plans (such as unit supports, unit availability, plan time and so on) in the year 1992 may be changed for the year 1993. Consequently, a plan constructed in 1992 with respect to the 1992 formal constraints (as given by degree regulations) may become inconsistent with the 1993 constraints. The 1992 plan is thus minimally revised in 1993 such that it is consistent with the 1993 formal constraints. This revision may require relaxation of 1992 preferential constraints if the 1993 formal constraints and the 1992 preferential constraints are together unsatisfiable.

3.3.14 Data Dependency

Some plan objects may be dependent on others through the support constraints. Because objects are chosen to satisfy certain constraints, these choices are considered as dependent on the respective constraints. Such dependencies are recorded as justifications for the objects in a plan, facilitating revision, explanation derivations and advisory tasks.

3.3.15 Experiential Knowledge

CDSP problems usually have a database of previous plans. The database may be used as a knowledge source for planning. A person may use information drawn from the database to formulate preferential constraints which would focus search for a plan.

To summarise, we recognise that CDSP problems are not amenable to just algorithmic solutions. They also require heuristic knowledge and reasoning with symbolic knowledge in the problem solving. In the CDSP model, we do not attempt to model the human cognitive processes in solving the problems. Instead, we develop a practical computational model for problem solving with theoretical basis.

A model for an automated environment for CDSP needs to adequately capture the various types of problem knowledge and the computational mechanisms as discussed

\[^2\text{Modeling the human cognitive processes in problem solving is a difficult problem and it is a major area of active research in AI. Some interesting work on cognitive modeling from the AI perspective is reported in [6,113,132].}\]
3.4. Existing Models

above. What is required is:

- an input language for stating a problem and serving as the communication link for the user to issue constraints to the system;

- a knowledge representation formalism for encoding and representing knowledge internally in the model;

- an intelligent problem solving machinery to address pragmatic concerns in CDSP problem solving, and make effective use of existing knowledge to find solutions efficiently; and

- a human-computer interface through which constraints are received by the computer, and results (and traces) of computations, explanations and advice communicated to the user.

The problem solving machinery has to reason with the variety of constraints to search for solutions. The user serves as an oracle for interactively guiding the search through the preferential constraints. The other functionalities of the machinery are:

- incremental planning,

- plan revision,

- detection and resolution of inconsistencies,

- conditional reasoning,

- advising the user and eliciting useful information from the user with respect to some existing body of knowledge, and of action to pursue,

- using experiential knowledge.

We now review some of the existing relevant problem solving models and discuss their inadequacies in addressing the CDSP problems.

3.4 Existing Models

CDSP problem solving draws upon the AI areas of planning and constraint satisfaction. Chapter 2 gives an account of theories and techniques developed in these areas which are relevant to CDSP problem solving.

In the following subsections, we critique the constraint satisfaction and planning models in light of the CDSP problems. We show that formulating complete CDSP problems in terms of the existing models is not feasible.
3.4.1 CDSP Problems as Constraint Satisfaction Problems

The CDSP problems require satisfaction of constraints. Hence, some formulation of the CDSP problems as constraint satisfaction problems (CSPs) is required in the CDSP problem solving model.

As discussed in Chapter 2, a CSP is characterised by a tuple of known sets: $<C, V, D>$, where $C$ is a set of constraints, $V$ is a set of variables and $D$ is a domain of values that the variables from $V$ can assume. A solution to a CSP is the assignment of values to the variables such that all the constraints in $C$ are satisfied.

Although the CSP model covers a broad and useful spectrum of problems, many real-world problems requiring constraint satisfaction cannot be adequately formulated in it. It is too simple a representation for CDSP problems. It is lacking in the following ways.

- CSPs assume that the constraints are static during problem solving. In CDSP problems, new constraints appear either through preferences or constraint propagation and hence require dynamic constraint processing. Any change to a CSP through changes in one or more of the sets of variables, domains and constraints, defines a different CSP problem\(^3\). This assumption is too strong for the CDSP problems. The CDSP properties require search for a solution in an incremental fashion drawing on the knowledge gained from the history of problem solving.

- In the CSP formulation of a problem, the number of variables are fixed but in the CDSP problems, the number of variables are usually not known prior to the start of problem solving. They get defined during the course of problem solving. For example, in degree planning, the number of courses that are to appear in the final plan is not known \textit{a priori}. They are defined as the planning progresses. Defining variables to assume values from plan objects is not trivial. It requires a clever casting of the problem such that the variables are defined prior to running the search. In degree planning, it can be done in terms of dummy variables but in general, it may not be possible.

- CSPs do not allow distinctions between constraints and their exploitation in problem solving. As seen above, several kinds of constraints pertain to the CDSP problems. Their characteristics and strengths determine their potential uses in directing search for solutions.

\(^3\)Petrie [124] calls this problem the \textit{problem formulation} problem and argues that it is the most serious limitation of CSPs.
3.4. Existing Models

- CSPs do not support dynamic exploration of solution space. The CDSP problems characteristically rely on user guidance in search and support exploration of solution space. The CSP machinery does not capture justifications for variable assignments nor maintain multiple solution possibilities to facilitate these aspects of problem solving.

- Revision (including minimal revision) is not supported in the CSP model. Contingencies arise in CDSP problems and they need to be addressed. They may occur as a result of preferential constraints or inconsistencies arising during problem solving. The CSPs address the contingencies by solving the problem anew. They lack the machinery to perform revisions on an existing solution. Any change in constraints, variables and values result in a new CSP problem and it is solved anew.

- CSPs do not support informed recovery from inconsistencies. In problem solving, situations often arise when there are no consistent solutions. In such cases, because the CSP formulation does not use a justification structure, it cannot point to the culprits and aid in deciding on actions to pursue in recovering from the inconsistency.

The CDSP problems can be formulated as CSPs but many of the desired problem solving features will be lost. In this formulation, a decision is made once, and is independent of other decisions. A CSP solution is usually not revised i.e., revision and defeasible reasoning are traditionally irrelevant to the CSPs. If the problem changes, it is solved anew. Each change is regarded as a new CSP problem. Also, in this formulation, reasoning is algorithmic with a single, global justification for the solution. It does not identify and use “islands” of previously reasoned knowledge to avoid re-computations and also to guide solution revision. The CSP formulation does not adequately capture the pragmatic features which characterise the CDSP problems, and needs to be enhanced for it to model the complete CDSP adequately.

3.4.2 CDSP Problems as Dynamic CSPs

In CSPs, each change lends a CSP to a different CSP. Any such change to some existing solution is not provided for in the CSP paradigm. The dynamic character of CDSP problems is a major divergence of CDSPs from CSPs.

Research work in the area of constraint satisfaction has mostly concentrated on static networks of variables and constraints [40,95,107]. In these networks, the variables
and constraints remain fixed. Although various CSP techniques (including backtracking and consistency-based approaches [38]) have been developed to determine variable assignments which result in a consistent network (a consistent network represents all solutions), few results have been reported in theories which effectively manage a constraint network when variables and constraints get added to and deleted from the network. Some of the results are reported in [64, 106, 137, 152].

Seidel's work [137] on incremental constraint satisfaction methods only allows handling of a changing set of variables. Freeman-Benson et al [64] have presented an incremental constraint solving theory for maintaining an evolving solution to a set of constraints labeled with their respective strengths (called a constraint hierarchy) when constraints and variables are added and removed. A spectrum of algorithms have been developed for variants of this theory. The theory provides an expedient approach to maintaining an evolving solution to a constraint hierarchy by making only minor changes to an existing solution. This approach is claimed to be better than solving a constraint hierarchy from scratch each time a change is made. However, this approach is only applicable to problems in which constraints can be put in a hierarchy based on strengths. It does not address the issue of how constraint satisfaction is performed in a domain of diverse and complex constraints such as typified by CDSP problems. Nonetheless, it is an interesting research aspect to pursue in CDSP problem solving if the preferential constraints are defined hierarchically and once a constraint satisfaction problem solver is in place.

Mittal and Falkenhainer [106] have extended the notion of CSPs to model a special class of dynamic real-world problems. The extension is made by including constraints about variables (called activity constraints) which are considered in each solution. This is in addition to the compatibility constraints that are present in CSP. Through the activity constraints, conditions are expressed under which variables are and are not active, and in addition to the search performed by the CSP, inferences get performed about variable activity. The motivation for the theory was the key characteristic of the class of problems that “constraints on introducing and removing variables from a potential solution closely interact with constraints on consistent assignment of values to some already identified set of variables” [106]. This work is appealing for solving CDSPs provided it is enhanced to dynamically create variables and allow for changes in compatibility constraints. Also, while the support and incompatibility constraints can be viewed as the activity constraints, temporal reasoning for scheduling cannot be easily captured in this model.
Spencer et al [152] have applied truth maintenance to effectively manage a network of constraints in the setting of plan recognition in a temporal domain with just temporal constraints. Although CDSP problems very different to the problem of plan recognition, the attempt at modeling dynamic constraint satisfaction using the ATMS approach is relevant to the present work. It serves as a starting point for the IPLAN approach which uses an ATMS to capture an incremental and exploratory problem solving, developed for CDSP problems.

Research in dynamic constraint satisfaction is relatively new and have been domain specific. The theories which are relevant to DCSPs (as reported above) cannot be adequately used to model CDSPs. This is primarily because in CDSPs, the constraints, variables, and domains associated with each variable may get defined as problem solving progresses. Hence, the exploratory nature of problem solving is not supported in the existing theories. The properties of CDSPs motivate extensions to the existing CSP theories to adequately capture dynamic constraint satisfaction as required by CDSP problem solving. These extensions culminate in the theory for CDSP problem solving.

3.4.3 CDSP Problems are Atypical Planning Problems

Planning has been restrictively defined by AI researchers (such as Nilsson [120], Sacerdoti [135], Georgeff [71] and others) as the formulation of actions and their aggregation into sequences which when executed by agents would lead to desired goals. It is perceived as an activity requiring temporal reasoning and reasoning about actions. Most research in the planning area has concentrated on planning with this view.

CDSP problems involve a kind of reasoning that does not require search for fine-grained action sequences, such as are required to guide a robot to achieve some goals. Rather, it requires search for a compatible set of variable assignments to objects which may not necessarily be temporal objects. Also, the kinds of constraints in the CDSP problems are different. Hence, CDSP problems do not strictly conform to the classical definition of planning. They characterise a special class of “planning” problems with their own characteristic properties.

Wilkins’ SIPE [167] hierarchical action planner is one of the most influential contributions to AI planning. It improves on previous planning formalisms in its support of both automatic and interactive generation of plans, extensive use of constraints and resources, efficient methods for representing properties of objects, specification of plan rationale, and ability to express deductive rules for deducing the effects of actions.

SIPE uses a simple and straightforward constraint satisfaction algorithm. Con-
Chapter 3. CDSP Model

Constraints are used to filter possibilities at different levels of the search hierarchy. It is interactive—a major improvement on previous planners. Interactive planning is invoked through a constraint language.

Importantly, SIPE has been used to solve practical problems being successfully used to develop commercial prototypes for industrial environments [168]. SIPE is an ongoing project and few results (including the internal workings of the system) have been reported on the work already completed.

From the reported work [167,168], SIPE, in its present form, is not suitable for the CDSP class of problems because these problems do not conform to the properties of hierarchical planning. It also does not support most of the other CDSP properties. The constraint language in SIPE does not cater for the different kinds of constraints of CDSPs, incremental planning, conditional query processing, learning from experiential knowledge, use of knowledge contained in constraints in directing search and user-guided exploration of solution possibilities.

In general, because CDSP problems differ in properties from fine-grained action sequence planning problems, existing planning techniques are inadequate for the CDSP problem solving. Typically, AI planners reason with time and action contingent upon world states. They do not support dynamic reasoning with a variety of both fixed and preferential constraints as required for CDSP problems and the desired features of incremental planning, plan revision and user-guided exploration of solution possibilities.

3.4.4 CDSPs are Atypical OR Problems

Reasoning with diverse kinds of constraints in CDSP problem solving makes CDSPs different to known standard problems.

Although Operations Research (OR) establishes theoretical guarantees such as completeness and optimality, the technique is not suited to the human style of problem solving which the AI-based CDSP model attempts to capture. The heuristic and implementational aspects are lost in the OR formulation. Dhar and Ranganathan [45] compare and contrast the expert system and integer programming (IP) formulations of constraint satisfaction problems in an experimental domain. They demonstrate several drawbacks with the IP model which are addressed here for CDSP problems.

CDSP problems cannot be formulated as OR problems because the OR techniques cannot express the variety of quantitative and qualitative constraints\(^4\), cannot revise

\(^4\) In an integer programming formulation of the CDSPs, the knapsack and logical constraints destroy the integer solution property.
3.4. Existing Models

solutions and are inefficient (as found by Dhar and Ranganathan for a CSP [45]). The OR formalisation requires optimisation of an objective function which cannot be globally modeled for the CDSPs. Also, the OR formalisation assumes a fixed number of variables which is atypical of the CDSPs, and the formalisation hides the problem features, such as heuristics, which can be exploited in the search for solutions.

In general, academic mathematical models cannot be developed for the CDSP class of problems because these problems are ill-structured. The random nature of constraints, preferences and the complexity of problems cannot be adequately captured in mathematical models. These models would lack the heuristic and pragmatic features.

3.4.5 Constraint-Based Problem Solving Languages

There have been several attempts at modeling real-world problem solving which require some sort of constraint satisfaction. These attempts were reviewed in Chapter 2.

Most language-based systems that have been developed to solve classes of problems have been based on their narrow class-specific characteristics. None of the existing languages adequately capture the CDSP problem solving characteristics. Hence, CDSPs cannot be adequately formulated in these languages.

REF-ARF [54] and ALICE [92] are general problem solving languages built on high-level languages. These languages have been developed in attempts to improve upon the OR methods for solving combinatorial search problems which require constraint satisfaction. In these problems, the domains have to be extensional and the variables are known prior to problem solving. The languages also do not support the user facilities such as user-guidance of problem solving through dynamic addition of constraints, exploration of solution space and so on. They lack the expressive power to capture and solve the CDSPs.

CHARME [136] and PECOS [42] are constraint logic programming languages. They have been developed as extensions to PROLOG by embedding the constraint satisfaction techniques. Both of them solve a family of OR (scheduling and resource allocation) problems. In these languages, the problems and the constraints are declaratively stated, and procedural abstractions of mechanisms for satisfying the constraints are provided as the primitives. Again, these languages do not address the interactive, exploratory and incremental features in problem solving as required in CDSP problems. Also, CDSPs cannot be adequately expressed and solved using these languages as the languages do not have the expressive power to capture CDSPs.

None of the above languages support the exploitation of constraints in search. They
do not support the CDSP characteristics of dynamic constraint satisfaction, solution revision, interactive and incremental problem solving, solution revision, conditional queries and user-guided exploration of solution possibilities, replanning and advice generation.

3.5 Overview of CDSP Problem Solving Framework

Much research in the fields of AI and Mathematics is expended in searching for computational models with theoretical underpinnings for different classes of real-world problems. The models are developed in light of the problem structures and characteristics. Modeling is usually perceived as an experimental science.

In the remainder of this thesis, a computational model, called the CDSP model, for solving the CDSP class of problems is developed. The need for the model is motivated by the inadequacies of the existing models in capturing and solving the general class of CDSP problems. The proposed model combines and extends the existing problem solving theories to adequately capture the problem structure, and to support intelligent problem solving as required by the CDSP problems. Its objective is to capture problem solving in the CDSP class of problems by abstracting and formalising human expert knowledge so that it is amenable to mechanised reasoning.

3.5.1 Problem Statement

For a CDSP, the input is made up of a set of formal constraints Form-Cons, a set of preferential constraints Pref-Cons, a set of plan objects Plan-Objs, an existing plan Plan (if any), and any other available knowledge sources KS. The objective of CDSP problem solving is to use the given knowledge and the user as a guide, to draw up a plan of objects which satisfies all the constraints in $\Delta$. The role of the user in communicating his/her preferences to the system in guiding the planning process by exploring the possibilities and aiding in inconsistency resolution forms an important and integral part of CDSP problem solving.

There are two problem formulations with distinct problem solving characteristics identified and developed for CDSP problems. The first one treats CDSP problems as traditional CSPs but extends the CSP theory with integration of constraints in directing search. Here, any change to an existing CDSP problem is viewed as a new CDSP problem and solved anew. This formulation is called the Model-Search Problem. The other formulation treats CDSP problems as dynamic constraint satisfaction problems and addresses the essential features of user-directed exploratory and incremental prob-
lem solving. Here, the properties of constraints are also exploited in directing search. This formulation is called the IPLAN Problem. Formal characterisations are given for both of these formulations in Chapters 6 and 7 respectively.

Here, we give a formal statement of a general CDSP problem. A CDSP problem can be captured as a 5-tuple:

\[ P = \langle \text{Form-Cons}, \text{Pref-Cons}, \text{Plan-Objs}, \text{Plan}, \text{KS} \rangle \]

A solution, \( p \), to the above general CDSP problem formulation is an assignment such that all the constraints in \text{Form-Cons} and a maximal subset of constraints from \text{Pref-Cons} are satisfied. An assignment is a sequence of sets \( B_i \) of plan objects: \( <B_1, B_2, \ldots, B_i, \ldots, B_n> \). We call the \( B_i \)'s bins. The sets contained in a \( p \) are mutually exclusive. The justifications for inclusion of objects to \( p \) is explicitly maintained outside of the plan structure.

CDSP problem solving requires search for a \( p \) from a typically combinatorial set \( \mathcal{P} \) of plan possibilities through the application of operators/macro-operators from \( \Gamma \). Some of these operators are constraint satisfy functions. They make selections of plan objects and their supports, based on some existing state knowledge and information contained in selected constraints, and add them to the information contained in the existing state to get a new state. A goal state is reached when all the constraints evaluate to \true in the state i.e., when a schedule exists for the cumulative structure in the state exists.

3.5.2 CDSP Problem Structure

Like any other planning or constraint satisfaction problem, a CDSP problem is essentially a search problem. There are two components to the search process as given below.

- Selection—systematic use of inclusion constraints to select plan objects from a set of plan objects such that all the constraints are satisfied.

- Scheduling—assignment of the selected plan objects to a pre-defined sequence of bins such that all scheduling constraints are satisfied.

In the search, an informed selection-scheduling cycle continues until a schedule is found.

Both the components require constraint-based reasoning. From typically large solution spaces of CDSP problems, the use of constraint satisfaction metaphor results in pruning the spaces systematically using the constraints as filters, until a solution is
converged upon. Figure 3.3 gives a schematic representation of the general constraint satisfaction problem solving model. Conceptually, an intersection of sets of possibilities which satisfy each of the constraints results in a solution. Finding all the possible solutions is utterly impracticable. Hence, an informed problem solving theory which uses the constraints in directing the search is required. Opportunistic selection of constraints and their satisfaction characterise human constraint-based problem solving. Thus, in the selection process, constraints are used to direct search so that a solution is efficiently arrived at.

The scheduling component of CDSP planning shares characteristics with the traditional class of scheduling problems [3, 7, 69, 163]. It is like the bin packing problem [69] which involves packing a known number of objects in as few bins as possible such that “size” restrictions on the bins are met. In CDSP planning, given a known sequence of \( n \) bins, objects are placed in them in some systematic way such that the objects selected are consistent with the temporal constraints and the bin constraints. In particular,
3.5. Overview of CDSP Problem Solving Framework

the CDSP scheduling is an instance of the profile scheduling problem which has been extensively studied in the recent years by Bruno [18] and Dolev et al [47,48] and others.

The placement of objects in the profiles may involve ordered placements of other objects in the profiles as required by the support constraints. Through the support constraints, addition of certain plan objects may demand addition of further constraints and certain other plan objects to the plan in some order of precedence. This may give rise to overlapping or different multiple supports to the same plan objects, and consequently, adds complexity to the problem solving. The support constraints give the precedence constrained scheduling [69] character to the problem. The heart of the profile scheduling problem is precedence constrained scheduling and has been shown to be NP-complete [47]. A profile “is a sequence of natural numbers specifying how many processors are available at each time slot” and “a profile schedule is a partitioning of all the tasks into a sequence of sets which does not violate the precedence graph” [48].

As an illustration, consider a plan structure \( p = \langle B_1, B_2, \ldots, B_i, \ldots, B_n \rangle \). The addition of a certain object \( O_{ij} \) (where \( O_{ij} \) is the \( j^{th} \) object in the \( i^{th} \) bin) to \( p \) may require addition of certain other objects \( O_{kj} \) where \( k \) has to be from \( 1..i \). The restriction on object placement comes from the required precedence ordering of objects in the support structures. If an object \( O_a \) has a support \( O_b \) and \( O_a \) is available and placed in \( B_4 \) then \( O_b \) can only be placed in one of \( B_l \) bins where \( l \leq 4 \). The co-requisite support constraint of an object \( O_{ij} \) forces placement of the objects stated in the constraint in \( B_m \) where \( m \leq i \).

In CDSP problems, there are usually multiple, alternative prerequisite supports for a plan object. One of these is sufficient to support the object in the plan. Figure 3.4 depicts a plan structure with conjunctive supports for plan objects.

In the scheduling for CDSP problems, the task is to systematically fill up the sequence of profiles (or bins) with plan objects and their supports (as shown in Figure 3.4) so that all the local and global constraints are satisfied. The support structures dictate the distribution of plan objects into bins. Use of sophisticated data structures make search for a schedule efficient.

Together, the selection and scheduling components of CDSP problem solving requires search for a plan which satisfies a given set of constraints from a discrete finite and typically combinatorial space of alternatives. To generate (or guess) and test each possibility is a computationally intractable task. CDSP problem solving comprises of the Boolean satisfiability, bin packing and precedence constrained scheduling subproblems. All of these subproblems are known to be NP-complete (i.e., there exists no general
and efficient algorithm—not requiring exponential time—for solving them [69,163]). Hence, the CDSPs are very persuasively, but not conclusively NP-hard.

### 3.5.3 CDSP Problem Solving Language

CDSP problem solving is modeled in terms of a general problem solving language dedicated to the class of problems. The language we develop is called CL-IPS. CL is the constraint language in which a CDSP problem is encoded declaratively and input to the system. The constraints are well-formed sentences in CL. The IPS component is the “intelligent” problem solver which receives the problem encoded in CL and invokes an appropriate sequence of computational mechanisms to search for one or all solutions.

CL-IPS comprises of three components: an input language, a knowledge representation scheme for encoding and representing the domain and problem solving knowledge, and a collection of computational mechanisms to search for a satisfactory solution among a large number of possibilities.

Figure 3.5 shows the different components of CL-IPS. The input knowledge is interpreted and represented in perspicuous knowledge structures which facilitate reasoning with the knowledge contained in them. The knowledge structures are the objects on
3.5. Overview of CDSP Problem Solving Framework

Figure 3.5: Components of a General Problem Solving Language.

which the problem solving mechanisms operate.

3.5.4 CDSP Problem Solving Architecture

Figure 3.6 depicts the top-level general problem solving architecture that is proposed for the CDSP class of problems. This architecture supports all the features required for the CDSP problem solving.

The background knowledge is all the knowledge which is known prior to problem solving and it remains static during problem solving. The problem knowledge, which comprises of the the user preferences (dynamic knowledge) and the background knowledge, is encoded in CL and communicated to the problem solver. The intelligent problem solver draws upon a given pre-defined repertoire of operators to find a solution intelligently and efficiently, by processing the various kinds of knowledge which the problem solver has at its disposal. The user drives and guides the problem solver through CL. The experiential knowledge (from the knowledge source) is also exploited by the problem solver in guiding the search for a solution.

The interactive facility provides the essential human dimension to the problem solver. It allows the users to input their preferences to the problem solver and supports the user exploration of the solution space through formulation of queries (and conditional queries) as constraints in CL and communicating them to the problem solver. If there is a need for the user to seek advice on which actions to pursue in searching for a solution, the advisor component generates an optimal advice sequence based on the existing problem solving state and communicates it to the user through the advisor language.

All the knowledge pertaining to the problem solving is transparent to the user.
through user queries in CL. This includes the static background knowledge and knowledge in problem states. The information is retrieved and displayed by the browser.
Chapter 4

Knowledge Structures and Methods

4.1 Introduction

In this chapter, the knowledge formalisation and representation aspects of the general problem solving language, CL-IPS, for CDSP problem solving is presented. The chapter begins by giving an overview of knowledge representation concerns in reasoning systems. Then, it briefly presents an object-oriented knowledge representation language (called ORL) for use in the CDSP problem solving model. ORL is a small frame-like language (using object structures as the underlying data structures) adequate for the encoding and internal representation of CDSP knowledge. We illustrate the use of this language in representing CDSP knowledge (including constraints) as objects. The terminology of state-space problem representation is then introduced and knowledge structures which naturally pertain to CDSP problems and which facilitate CDSP problem solving are identified in terms of this representation. Computational procedures for deriving these structures and for accessing information from them are defined.

4.2 An Overview of Knowledge Representation

Reasoning systems derive their power from knowledge. The heart of any reasoning system is the knowledge it contains, and its strength depends on the richness of representation of the knowledge. An effective representation of knowledge is therefore generally considered to be the keystone to the success of reasoning systems [113]. Knowledge representation (KR) is an important concern in reasoning systems.

The various kinds of knowledge (namely the domain a system has knowledge about, and the task the system is to perform with this knowledge) need to be formalised, encoded and represented in a formalism which facilitates intelligent problem solving. Reasoning systems stress the need for the availability of expert knowledge in the systems
The arrows show flow of information.

Figure 4.1: Knowledge Representation in a Reasoning System.

along with associated knowledge handling facilities [108].

The basic problem of KR is the development and use of sufficiently precise notation which captures and encodes into knowledge structures descriptions, relationships (structural and behavioral), objects, goals, procedures, functions, processes, and other knowledge pertaining to the problem domain. The raw knowledge in the domain is formalised prior to the representation. Figure 4.1 summarises the steps in KR for a reasoning system. These steps are used in the formalisation and representation of problems in the CDSP domain.

The presence of various different kinds of knowledge and reasoning mechanisms make the KR task complex. It is essential that the KR formalism adopted meets the basic criteria of expressive power, understandability and accessibility [53]. The KR formalism must have sufficient expressive power to state, perspicuously and succinctly, both the declarative ("what is") and procedural ("how to") knowledge—referred to as the epistemological adequacy [99]. Rich [131] argues that a good KR formalism is one that possesses the following four properties.

i). Representational Adequacy—the ability to represent all of the various kinds of knowledge that is needed in the problem domain.

ii). Inferential Adequacy—the ability to manipulate the representational structures in such a way as to derive new structures corresponding to new knowledge inferred from the old knowledge.
iii). **Inferential Efficiency**—the ability to incorporate into the knowledge structure additional information that can be used to focus the attention of the inference mechanisms in the most promising directions.

iv). **Acquisitional Efficiency**—the ability to acquire new knowledge easily and efficiently.

Three basic representation formalisms, namely *frames* [105, 128], *production rules* [83], and *logic* [81, 119, 151], have emerged to support the complex task of representing various kinds of knowledge (including reasoning knowledge) for a problem domain in a knowledge-based system. They can be used to represent both the declarative and procedural types of knowledge\(^1\). In many AI systems, a hybrid of these formalisms have been used.

Various kinds of knowledge can be represented by the frame formalism. It is suited for KR in CDSP problems for various reasons as explained in [141]. In the following paragraph, we briefly describe the frame formalism and in the later sections on KR, we present the frame-based, ORL language, and demonstrate how ORL is used in representing CDSP knowledge.

The frame formalism originated from the work of Minsky [105]. It provides a method of combining declarations and procedures within a single knowledge representation environment. The fundamental organising principle underlying the frame formalism is the organising and packaging of knowledge. All the knowledge is partitioned into discrete knowledge structures (called *frames*) having individual properties (called *slots*). Frames can be used to represent broad concepts, classes of objects, or individual *instances* or components of objects. They may have slot values which are facts, rules or computational procedures. Also, they support organisation of knowledge into hierarchies or networks for property inheritance and default reasoning. They are especially suited to applications in which related pieces of knowledge need to be packaged and represented in a modular fashion. Fikes and Kehler [53] demonstrate how frame-based systems can be used to incorporate a range of inference methods common to both logic and rule-based systems.

---
\(^1\)Fikes and Kehler [53], and Friedman [67] review the available KR formalisms and assess their suitabilities for knowledge representation in reasoning systems.
4.2.1 CDSP Knowledge

The CDSP problem solving involves several kinds of knowledge. They are broadly categorised below.

i). Background knowledge.

Background knowledge is all the knowledge known prior to problem solving and which remain static during problem solving. It includes objects and formal relations over them, facts, rules, formal constraints, partially completed plans, plan objects, domain heuristics, learned knowledge structures and computational procedures.

ii). Control knowledge.

The control knowledge organises, guides and performs the problem solving process. This knowledge is usually abstracted as problem solving operators which are modeled as functions.

iii). Reasoned knowledge.

The reasoned knowledge comprises knowledge derived and accumulated during problem solving. This also includes the preferential constraints.

These pieces of knowledge need to be organised, packaged and represented in a manner that facilitates reasoning with them. We adopt the frame formalism for CDSP knowledge representation for its strengths and versatility.

4.2.2 ORL Representation of CDSP Knowledge

ORL² is a hybrid of the frames, rules and logic formalisms. It is a hybrid knowledge representation knowledge, combining frame, logic and production rule formalisms, in the tradition of KEE™ (for Knowledge Engineering Environment™)³. For the purposes of the CDSP model, a subset of features from KEE as realised in ORL is adequate.

²The ORL formalism is an enhancement of the R2L formalism presented in [141].
³KEE and Knowledge Engineering Environment are trademarks of IntelliCorp.
4.2. An Overview of Knowledge Representation

4.2.2.1 An Overview of ORL Formalism

The basic structure in ORL is an object. (Table 4.1 shows an ORL object schema). The attributes of the objects are represented as the object slots with qualifiers for checking the consistencies of the values assigned to the slots against the pre-defined possible values the slots can assume. The attributes can be facts, logic expressions, rules or procedural knowledge. Class properties are inherited through the isa links. Any meta-information about objects can also be packaged in the object as slots. Knowledge base transactions are modeled as functions on the objects. (The basic syntax of an ORL object is defined in Table 4.2.)

Some of the reasons for choosing a hybrid representation for CDSPs are explained below.

- It allows both declarative and procedural representation of knowledge in a single environment. This feature helps in representing constraints which are usually both declarative and procedural.

- It provides a concise structural representation of an object or class of objects.

- It has the ability to define a taxonomy of objects where objects lower in the taxonomy inherit information from higher level objects - through “property inheritance”.

- It allows modularisation and packing of the various kinds of knowledge—object attributes, procedures, active values, object links, logic statements, and rules—in a single environment.

- It supports efficient checking for consistency (such as conflicts and redundancies) and completeness in data through the qualifier functions attached to the slots.

- It supports restriction of values for slots by allowing specification of constraints which are to be satisfied by the slot values.

<table>
<thead>
<tr>
<th>isa (qualifier_0):</th>
<th>Object-Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute_1 (qualifier_1):</td>
<td>value_1</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>attribute_n (qualifier_n):</td>
<td>value_n</td>
</tr>
</tbody>
</table>

Table 4.1: An Object Schema in ORL.
• It supports representation of default knowledge.

In the ORL representation, related knowledge is packaged as objects in the same environment. An accessor and a modifier function is established for each slot in each object. They have the syntax: (get-attr object-id attribute-name) and (set-attr object-id attribute-name attribute-value) respectively. All the attributes defined for the objects have predefined types and each slot in an ORL object has a type checking function.

### 4.2.2.2 CDSP Knowledge Representation

In the CDSP domains, sets of attributes and sets of possible values (and any formal relations over them) for each of these attributes are usually known or computable through some known mechanism. Constraints are defined in terms of these sets and any formal relations over them. The procedural and functional abstractions of computational knowledge are known for each category of constraints. In reasoning with the constraints, a computational abstraction is inherited from an appropriate class and used.

A plan object structure is illustrated in Table 4.3. $AT_i$ and $ATV_i$ denote arbitrary attributes and sets of their possible attribute values respectively. Constraints are also represented as objects. A constraint object captures a declarative statement in CL of the constraint, an interpreted statement of the constraint, an operator to check whether the constraint is satisfied for some given plan, and operators to find ways of satisfying it, if it is already not satisfied, by inducing changes to an existing plan. Table 4.4 illustrates a constraint object.

Knowledge clusters as derived from experiential knowledge are represented as structures of sets of objects in ORL. These structures characterise generalisation/specialisation hierarchies in tree structures. The learned knowledge, its representation and exploitation are presented in detail later.
4.2. An Overview of Knowledge Representation

<table>
<thead>
<tr>
<th>Isa:</th>
<th>Plan-Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id:</td>
<td>&lt;a unique identifier&gt;</td>
</tr>
<tr>
<td>Weight:</td>
<td>&lt;a numeric weighting on the object&gt;</td>
</tr>
<tr>
<td>Prereq:</td>
<td>&lt;an AND/OR expression of prerequisite constraints&gt;</td>
</tr>
<tr>
<td>Co-Req:</td>
<td>&lt;a set of plan objects which are co-requisites of current object&gt;</td>
</tr>
<tr>
<td>Incomp-Obj:</td>
<td>&lt;a set of plan objects&gt;</td>
</tr>
<tr>
<td>Avail:</td>
<td>&lt;temporal restriction on plan object availability&gt;</td>
</tr>
<tr>
<td>$AT_i$:</td>
<td>$&lt;ATV_i&gt;$</td>
</tr>
</tbody>
</table>

Table 4.3: A Plan Object in ORL.

<table>
<thead>
<tr>
<th>Isa:</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label:</td>
<td>&lt;automatically generated unique identifier for the constraint&gt;</td>
</tr>
<tr>
<td>Type:</td>
<td>&lt;formal or preferential&gt;</td>
</tr>
<tr>
<td>Text-form:</td>
<td>&lt;constraint as expressed in constraint language&gt;</td>
</tr>
<tr>
<td>Interp-form:</td>
<td>&lt;internal representation of constraint&gt;</td>
</tr>
<tr>
<td>Source:</td>
<td>&lt;source of the constraint&gt;</td>
</tr>
<tr>
<td>Tester:</td>
<td>&lt;method to evaluate the constraint against some given state&gt;</td>
</tr>
<tr>
<td>Satisfier:</td>
<td>&lt;method to apply the constraint as a generator to some state&gt;</td>
</tr>
</tbody>
</table>

Table 4.4: A Constraint Object in ORL.
4.3 State-Space Terminology

Here, we introduce the graph-theoretic terminology pertaining to state-space search. This is used later in the derivation of knowledge structures and in the semantics of CL.

A state-space search representation is a tuple:

\[ < S, \Gamma, \mathcal{K} > \]

where
- \( S \) is a set of problem states which capture various kinds of state knowledge,
- \( \Gamma \) is a set of transition operators which modify the knowledge in \( S \) to produce a new state; these operators are modeled as functions, \( \gamma : S \rightarrow S, \gamma \in \Gamma \), and
- \( \mathcal{K} \) is a control strategy that decides at any given time in the search which operation to apply to the knowledge in some existing state.

State-space search is aptly captured in graph-theoretic terms. Figure 4.2 depicts a simple state space. The nodes (or vertices) represent the states from \( S \) and the arcs (or links) represent the operators from \( \Gamma \). \( S_0 \) is the root node. Its successors (or children) are \( S_1, S_2 \) and \( S_3 \), and \( S_0 \) is called the parent node of its successors.

The search process involves computation of successor nodes from parent nodes. If a single node \( S_{i+1} \) is computed from \( S_i \) then \( S_{i+1} \) is said to be generated and \( S_i \) is said to be explored. \( S_1 \) is a generated node of \( S_0 \).

A node \( S_i \) is expanded if all the successor nodes of \( S_i \) are generated. In Figure 4.2, \( S_0 \) expands to \( S_1, S_2 \) and \( S_3 \). \( S_2 \) expands to \( S_4 \) and \( S_5 \), \( S_3 \) expands to \( S_6 \), and \( S_4 \) expands to \( S_7 \).

A search procedure (or control strategy) organises the search process by determining the order in which the nodes are to be explored. It chooses states from each of the expansions until a solution is found. Usually, a solution is a path through the state.
4.4 Knowledge Structures and Methods

In this section, we present the knowledge structures and methods for constructing them. The knowledge structures capture the plan objects and their support structure succinctly capturing support knowledge. The semantics of the CL language (presented later in this chapter) and the search models (presented in the following chapters) are given in terms of these knowledge structures and the accompanying methods for deriving them. The knowledge structures facilitate problem solving by explicitly capturing support knowledge in forms which support efficient reasoning with it. In the search process, in particular, the plan object support structures which arise from the selection process are explicitly retained and rather than generating a schedule and testing it for satisfaction of the support constraints, the captured support structures are exploited in guiding the scheduling process. These structures satisfy the support constraints per se. Finding a schedule that satisfies the scheduling constraints, if one exists, from these structures is computationally more tractable. In addition to facilitating the scheduling process, the explicit retention of support relations serve as justifications for presence of objects in a schedule and supports derivations of explanations to queries on reasons for object presence in the schedules.

We begin by defining the composition of a state in the problem space. Some of the knowledge structures are derived using information captured in these states. Also, we later give semantics to CL in terms of these states. The various knowledge structures which characteristically emanate from CDSP problem domains due to support relations between objects are identified and their properties presented. Finally, we show how the properties of these structures are exploited in tractably finding a schedule.

4.4.1 Problem Solving State

CDSP problem solving is state-based. The composition of states is dependent on the control strategy used by the search mechanism adopted—for example, as we shall see in Chapters 5 and 6, in formulations of different approaches to CDSP problems as state-based search, different kinds of knowledge are captured in states. We define a
state\textsuperscript{4} in CDSP problem solving as:

\[ S_i: \langle AC, C, EX, X, CS, PC, schedule \rangle \]

where
- \( S_i \) represents the \( i \)\textsuperscript{th} state,
- \( AC \) is a set of constraint statements given in CL,
- \( C \) is a constraint picked for satisfaction from \( S_i \) and is satisfied in \( S_{i+1} \),
- \( EX \) is a set of plan objects chosen, which together with \( S_{i-1}.CS \) satisfies \( C \),
- \( X \) is a conjunctive support structure (see §4.4.4) for \( EX \),
- \( CS \) is a cumulative structure formed from conjunctive support structures,
- \( PC \) is a sequence of \textit{processed} constraints, and
- \textit{schedule} is a sequence of temporal intervals (or bins) containing objects from \textit{Plan-Objs} \( \cup \text{CONST} \), as defined later in §4.4.9.

\( \Delta \) is a set of both formal and preferential constraints that is known prior to search and \( AC \) is a set of constraints accumulated during search. \( AC \) depends on the search path pursued.

The above formulation forms the underlying control structure for CDSP problem solving. The states succinctly capture all the relevant problem solving information in them.

\subsection{4.4.2 Object Supports}

The object supports are the co-requisite and prerequisite relations between objects (see Chapter 3, page 33 for the properties of support relations).

A support structure, \( SStruc \), of a plan object \( O \) is a kind of a directed acyclic graph (\textit{dag}). The nodes are plan objects or constraints, and an arc captures the prerequisite and co-requisite support relations. \( SStruc \) recursively captures the plan objects and all their supports (constraint objects do not have supports). The terminal nodes are either plan objects which do not require supports or constraints.

Figure 4.3 depicts a typical \( SStruc \). The \( O_s \) and \( C_s \) denote arbitrary plan and constraint objects respectively. The solid lines denote prerequisite supports whereas the dotted line denotes co-requisite support. Some prerequisite relations from the figure are \( C_1 \prec O_3 \prec O_2 \prec O_1, O_4 \prec O_2 \prec O_1, \) and \( C_2 \prec O_9 \prec O_2 \prec O_1 \) (where \( \prec \) represents

\footnote{In this thesis, all state information from some given state, \( S_i \), is referenced by \( S_i. \text{<attribute>} \) such as \( S_.AC, S_.C, S_.EX, S_.X, S_.CS, S_.PC, \) and \( S_.schedule \) for the \( i \)\textsuperscript{th} state definition.}
prerequisite support). $O_6$ has $O_7$ as its co-requisite support, written $O_6 # O_7$. Note throughout the thesis, the edges of the graphs are assumed to be directed upwards and dags are always represented as trees.

The support structures $SStruct$ are created from support information given for plan objects. The following function retrieves the support structure for a plan object.

\[
\text{get-support: Plan-Objs} \rightarrow \text{Set-Of SStruct.}
\]

For example, in degree planning, given the prerequisites for a unit Comp3013 as Comp2012 and (Sum-Wts (Mathematics Statistics) B At-least 1), its support structure is given by a set of tuples: \{\{(Comp3013 Comp2012) and (Comp3013 (Sum-Wts (Mathematics Statistics) B At-least 1))\}\}.

### 4.4.3 $\Pi$ Structure

$\Pi$ is a structured data object which combines the support structures for a given set of plan objects.

Given the support structures for two objects, $O_i$ and $O_j$ (for arbitrary $i, j$ and $i \neq j$), the $\Pi$ structure for \{\{\{O_i, O_j\}\}\} is the combination of the two support structures, written

\[
\Pi_{\{O_i, O_j\}} = SStruct_{O_i} \cup SStruct_{O_j}.
\]

The object support structures form the building blocks for $\Pi$. Figure 4.4 illustrates the combination.

---

5 Recall from Chapter 3: given two distinct objects, $O_i$ and $O_j$, if $O_i # O_j$ then it does not necessarily mean that $O_j # O_i$.

6 Note that, in this thesis, algorithms are given for the important functions and only signatures are given for the less important functions.

7 In the signature declarations for functions, Set-Of and $P$ are used to denote sets and power sets respectively.
A $\Pi$ is created for some given set of plan objects through the $get-pi$ function:

$\textit{get-pi}: P \text{ Plan-Objs} \rightarrow \text{Set-Of} \ \Pi$.

It uses the $get-support$ function.

### 4.4.4 Conjunctive Support Structures

$\Pi$ is an AND/OR structure and hence embodies multiple support structures for the combined set of objects at the root node. Each of these structures is a \textit{conjunctive support structure}.

In the planning process, each of the constituent conjunctive support structures embedded in $\Pi$ need to be extracted and considered. The \textit{disjunctive normal form} (dnf) transformation of a $\Pi$ does not adequately re-represent all the desired information from $\Pi$. It yields a set of “flat” disjuncts which loses the precedence information. It is vital that the precedence information is retained in an easily accessible form because it forms the constraints on the object scheduling process\(^8\).

Hence, in the fission of $\Pi$ we account for the object supports by explicitly labeling the nodes with the precedence information. An enhancement is made to the dnf conversion process by incorporating the structural information. This enhanced technique, called the \textit{disjunctive ordered form} (dof), takes a $\Pi$ and decomposes it into its constituent \textit{incompatibility-consistent}\(^9\) CSS structures preserving the precedence property.

This behavior is modeled by the following $dof$ function:

$dof: \text{Set-Of} \ \Pi \rightarrow \text{Set-Of} \ CSS$

where CSS is a set of conjunctive support structures. Disjuncts in a $\Pi$ structure result

\(^8\)Scheduling is an essential component in CDSP problem solving. It is discussed in detail in §4.4.9.

\(^9\)An \textit{incompatibility-consistent CSS} structure is one which contains objects that are not incompatible. Its test is explained on pages 89, 270.
in multiple CSS structures. The CSS structures are \( \Pi \) structures with no disjuncts.

Figure 4.5 demonstrates the fission of the \( \Pi \) of \( \{O_1, O_2, O_5\} \) into its conjunctive support structures, CSS\(_1\) and CSS\(_2\). \( O_1, O_2, O_3, O_4 \) and \( O_5 \) are arbitrary plan objects and \( C \) is an arbitrary constraint. \( T \) denotes the fusion point of the support structures of objects in \( T \) and forms the root node of \( \Pi \).

Computationally, given a \( \Pi \) with nodes given by the set \( V = \{v_1, \ldots, v_n\} \), and \( v_i \) and \( v_j \) as adjacent nodes on \( \Pi \), the \text{dof} \ function extracts the support relations between nodes of \( \Pi \) as tuples \( <v_j \# v_i> \) or \( <v_j < v_i> \), depending on whether \( v_j \) is a co-requisite support of \( v_i \) or \( v_j \) is a precedence support for \( v_i \) respectively. \text{dof} \ then converts \( \Pi \) into an AND/OR expression with \( (v_j \# v_i) \) and \( (v_j < v_i) \) structures as literals in the clause, and finds the \text{dnf} \ of the clause. The resulting disjuncts are sets of \( <v_j \# v_i> \) and \( <v_j < v_i> \) tuples which describe the CSS structures.

Below are the properties of the conjunctive support structures.
• The CSS structures are directed acyclic graphs which characterise intrees [69]. In these structures, objects at lower levels are supported by multiple objects at higher levels.

• The CSS structures are precedence/coincidence graphs with partial ordering between plan objects. The ordering satisfies prerequisite and co-requisite constraints on plan objects contained in the structures. The support ordering of the nodes in a CSS = (V, E) is a sequence < v₁, ..., vₙ >, such that ∀ < vᵢ, vⱼ > ∈ E, either vⱼ is a precedence support of vᵢ (given as < vⱼ < vᵢ >) or vⱼ is a co-requisite support of vᵢ (given as < vⱼ#vᵢ >).

• The root node in a CSS structure is arbitrarily marked as T.

• An important property is that each plan object in a CSS structure has a single conjunctive support.

As an example from degree planning, given the following prerequisite structures:

<table>
<thead>
<tr>
<th>Unit</th>
<th>Prerequisites</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp3013</td>
<td>Comp2012 and (Sum-Wts (Mathematics Statistics) B At-least 1)</td>
</tr>
<tr>
<td>Comp2012</td>
<td>Comp2011</td>
</tr>
</tbody>
</table>
| Comp2011 | Comp1002 and (Sum-Wts Mathematics At-least 2)  
or (Sum-Wts (Mathematics Statistics) At-least 2) |
| Comp1002 | Comp1001 or Comp1003                            |
| Comp1001 | nil                                              |
| Comp1003 | nil                                              |

The conjunctive support structures for the unit Comp3013 are:

1. ((T Comp3013) (Comp3013 Comp2012) (Comp2012 Comp2011) (Comp2011 Comp1002) (Comp1002 Comp1001) (Comp2011 (Sum-Wts (Mathematics Statistics) At-least 2)) (Comp3013 (Sum-Wts (Mathematics Statistics) B At-least 1)))

2. ((T Comp3013) (Comp3013 Comp2012) (Comp2012 Comp2011) (Comp2011 Comp1002) (Comp1002 Comp1003) (Comp2011 (Sum-Wts (Mathematics Statistics) At-least 2)) (Comp3013 (Sum-Wts (Mathematics Statistics) B At-least 1)))

3. ((T Comp3013) (Comp3013 Comp2012) (Comp2012 Comp2011) (Comp2011 Comp1002) (Comp1002 Comp1001) (Comp2011 (Sum-Wts Mathematics At-least 2)) (Comp3013 (Sum-Wts (Mathematics Statistics) B At-least 1)))
4.4. Knowledge Structures and Methods

4.  

$$((T \text{ Comp3013}) (\text{Comp3013 Comp2012}) (\text{Comp2012 Comp2011}) (\text{Comp2011 Comp1002}) 
(\text{Comp1002 Comp1003}) (\text{Comp2011 (Sum-Wts Mathematics At-least 2)}) (\text{Comp2011 (Sum-Wts (Mathematics Statistics) B At-least 1)}})$$

4.4.5 Addition of Conjunctive Support Structures

At each state, $S_i$, in the search, a conjunctive support structure (cs) is chosen from the set $S_i.CSS$ such that the combination of cs and $S_i.CS$ is schedulable.

We define a binary operator $\oplus$ as the combination operator:

$$- \oplus - : CSS \times CSS \rightarrow CSS.$$  

$\oplus$ is a commutative operator and it combines two given CSS structures according to the following rules.

i). If the two structures have the same sub-structures for each of the same nodes then the two structures are the same, and one of the structures is returned.

ii). If one of the structures is a sub-structure of the other then the larger structure is returned. This is the idempotence property of $\oplus$.

iii). There may be objects at terminal nodes common to both the structures. They do not require supports and hence, addition of the structures will not violate the unique support property of CSS. In such cases, the structures get added.

iv). If the two structures have some plan objects common to both the structures but their supports are different in the two structures then addition is not possible because the addition will violate the unique conjunctive support property of the CSS structures.

Figure 4.6 demonstrates the semantics of the $\oplus$ operator in each of the above first three cases.

4.4.6 $\wedge$ Structure

Scheduling is an essential component of the CDSP planning process. It sequences the objects from a given CSS structure into temporal intervals in which they are available (as given by the availability constraints). In the sequencing, the support relations between objects have to be preserved and all the scheduling constraints (viz., temporal constraints and weight constraints on each interval) have to be satisfied.
The support relations impose constraints on the scheduling of objects\textsuperscript{10}. In the CDSP model, search for a schedule is guided by the knowledge captured in the conjunctive support structures for plan objects. This knowledge is re-represented in structures which facilitate exploitation of the information in guiding search for a schedule.

Scheduling requires the generation of all sequence possibilities by assigning the plan objects in the intervals they are available, and searching for a schedule that is consistent with the local interval constraints and any (including user-specified) temporal constraints. Prior to the search for schedules, the CSS structures get re-represented as A structures in which independent supports for plan objects are succinctly captured. The overlapping supports for plan objects get combined. This representation facilitates the scheduling process without the loss of the ordering properties between the plan objects and their supports.

In the following subsections, we describe the transformation of the CSS structures to A structures, give the properties of the A structures and discuss how the information contained in them is used in the scheduling process.

\textsuperscript{10}As discussed in Chapter 3, precedence constrained scheduling is an NP-complete problem.
4.4. Knowledge Structures and Methods

4.4.7 Derivation of $\Lambda$ from $CSS$

For $\Lambda$ structure representations and properties see §4.4.8. The derivation of $\Lambda$ structures from the $CSS$ structures is modeled by the function:

$\text{get-supp-struct}: CSS \rightarrow \Lambda,$

where, $CSS$ is a conjunctive support structure, and $\Lambda$ is a set of $\Lambda$ structures.

Consider $CSS_1$ and $CSS_2$ from Figure 4.5 as illustrative examples. $\text{get-supp-struct}$ produces $\Lambda_1$ and $\Lambda_2$ respectively from these structures (see Figure 4.7), where

$\Lambda_1 = \langle\langle \{O_1, O_2\}, \{O_3, O_4\}, \{C\}>, \langle\{O_5\}\rangle, \rangle$, and

$\Lambda_2 = \langle\langle \{O_1, O_5\}>, \langle\{O_2, O_4, C\}\rangle\.\rangle$.

In $CSS_1$, there are two occurrences of $C$: one as a support for $O_3$ and the other as a support for $O_2$. The corresponding $\Lambda$, $\Lambda_1$, captures $C$ at level 2 because level 2 is the least upper level (lul) in which $C$ supports both $O_3$ and $O_2$. All the other objects remain at the levels in which they appear in $CSS_1$. Placement of constraints at the $luls$
constrain scheduling by restricting placement of the plan objects, which are selected to satisfy the constraints, at the lul or higher levels.

In CSS2, one instance of \(O_5\) forms a precedence support for \(O_1\) and the other does not form support to any object. The lul for placing \(O_5\) is level 1. In the scheduling of objects, \(O_5\) can be placed at any level at or above level 1.

In general, in the derivation of \(\Lambda\)s from CSS structures the following steps are followed in \textit{get-supp-struct}.

1. All overlapping structures rooted at \(T\) are identified and the CSS is broken up from \(T\) into non-overlapping sub-structures.

2. \textit{Breadth-first} traversal is performed on each of these structures to determine the sets of objects at each level resulting in transition \(\lambda\) structures (TLS).

3. Each of the TLS structures are configured to ensure they are consistent with the lul property resulting in \(\lambda\) structures.

4. Finally, the \(\lambda\) structures are ordered with respect to their lengths, from largest to smallest, resulting in the \(\Lambda\) structure.

In addition to constraining the scheduling process, the \(\Lambda\) structure facilitates search for a schedule and gives completeness to it. It is required that all the plan objects in a schedule are \textit{well-supported}. For example, in the scheduling of a \(\Lambda_j\), if an object, \(O\), is at position \(i\) in \(\Lambda_j\) then \(O\) can only be scheduled in the temporal intervals higher in the sequence than those intervals in which the objects in \(\Lambda_j\) at positions lower than \(i\) can be scheduled. Objects with no support restrictions may be placed anywhere in a schedule provided they satisfy the availability constraints (see §4.4.9). In degree planning, a third year unit may be taken in the first year provided it does not require a prerequisite. Likewise, a first year second session unit may be taken in the final year second session provided the unit is not required as a prerequisite support for some other unit in the plan. The \(\Lambda\) structure captures and facilitates use of these kinds of support knowledge.

4.4.8 Properties of \(\Lambda\)

The properties of the \(\Lambda\) structures are enumerated below.

1. A \(\Lambda\) structure, \(\Lambda_i\), is a sequence of sequence of sets:

\[\Lambda_i = \langle \lambda_1, \lambda_2, \ldots, \lambda_j, \ldots, \lambda_n \rangle,\]

where the \(\lambda\)s are sequences of elements from \(P\) (\(Plan-Objs \cup CONST\)).
2. The elements of a $\Lambda_i$ are mutually exclusive i.e., they do not have elements in common.

3. Each $\Lambda$ is ordered on magnitudes of its constituent $\lambda$s.
   Given some $\Lambda_i = < \lambda_1, \lambda_2, \ldots, \lambda_j, \ldots, \lambda_n >$,
   $\forall j : 1 \ldots n - 1$, $\text{length} (\lambda_j) \geq \text{length} (\lambda_{j+1})$.

4. Given a $\lambda: < B_1, \ldots, B_n >$, if an object is in $B_j$, where $1 \leq j \leq n$, then all its
   precedence supports are contained in $B_{j+1} \ldots B_n$, and all its co-requisite supports, if any, are contained in $B_j \ldots B_n$.

5. Given a $\lambda: < B_1, \ldots, B_n >$, objects appear at the lowest possible set in the sequence forming supports to objects in the preceding sets. If an object is contained in $B_1$ then it does not form support to any object in $\lambda$. On the other hand, if an object is contained in $B_n$ then the object does not need any support in the given $\lambda$.

6. The ordering in a $\lambda$ denotes $\lambda$-supports (denoted by $\preceq$) of sets of objects. In $\lambda_1$ of $\Lambda_1$ of Figure 4.7, $\{C\} \preceq \{O_3, O_4\} \preceq \{O_1, O_2\}$.

7. $\preceq$ defines a partial order on $B = \{B_1, B_2, \ldots, B_n\}$.

8. All the objects in a $\lambda$ are well-supported i.e, each $\lambda$ satisfies the support constraints on its objects.

4.4.9 Scheduling

In CDSPs, scheduling of objects requires placement of objects in a known number, $n$, of a temporal sequence of discrete time intervals preserving the support constraints (both precedence and coincidence constraints) between objects and satisfying all the scheduling constraints. The precedence scheduling problem, in general, is known to be NP-complete (see [47], and [163] for a proof). However, because most of the constraints in CDSP problems specify conditions in terms of explicitly and precisely defined temporal intervals, they give a higher degree of computational efficiency in search for a schedule. Only the user-specified precedent (i.e., “take-before”) and coincident (i.e., “take-concurrently”) constraints are relative and for these, explicit time-stamping is not possible. Although temporal reasoning required in the CDSP type scheduling is interval-based, the temporal ordering between the intervals are pre-defined and they remain static. These properties simplify the problems confronted in general modeling of temporal reasoning as identified by Allen in [5].
In CDSP scheduling, we use domain knowledge, and a knowledge structures and their derivatives to tractably search for a schedule for some given CSS structure.

In this section, we show how the proposed knowledge structures facilitate the search for schedules. We begin by giving a formal definition of a schedule.

A schedule for a precedence structure, CSS, in graph-theoretic terms, is a sequence of sets \(< SS_1, SS_2, \ldots, SS_i, \ldots, SS_n >\) such that:

i). \(\forall i = 1 \ldots n, SS_i \subset Plan-objs \cup CONST,\)

ii). the sets \(SS_i,\) for \(1 \leq i \leq n,\) partition the vertices of CSS.

iii). if \(x \in SS_i\) and \(y \in SS_j,\) for \(1 \leq i \leq j \leq n,\) then there is no path from \(y\) to \(x.\)

The scheduling process is modeled by the function:

\[
schedule: CSS \times P CONST \times P TI \rightarrow Schedules \cup \{fail\},
\]

where,
\(CSS\) is a set of conjunctive support structures,
\(CONST\) is a set of constraints,
\(TI\) is a set of temporal intervals,
\(Schedules\) is a set of schedules, and
each \(SS_i\) in each schedule is a subset of \(Plan-objs \cup CONST.\)

(Note we use \(SS\) to refer to bins in a schedule. It distinguishes them from bins in a plan.)

\(schedule\) takes a cumulative structure \(CS,\) a set of scheduling constraints and a set of non-committed discrete temporal intervals, and returns a schedule if one exists. Otherwise it returns \(fail.\) It uses the \(get-supp-struct\) to get the \(A\) structures from some given \(CSS.\)

A conjunctive support structure is \(schedulable\) if and only if a schedule can be found for the structure which preserves the support constraints on all the objects and satisfies all the scheduling constraints. It is \(non-schedulable\) otherwise. Also, it is sufficient to show a schedule does not exist for a \(CS\) structure if it can be shown that a subset of plan objects from the existing \(CS\) is non-schedulable. During search for a schedule, a schedule may be \(partially\) \(elaborated.\) This is so iff any of its constituent sets contains an element which is a constraint in implicit form. The schedule is \(fully\) \(elaborated\) otherwise. We assume, by default, that all partial schedules are schedulable unless on further elaboration, they are discovered to be unschedulable.

A fully elaborated schedule which satisfies all the problem constraints is a \(well-formed\) \(plan\) in reverse. That is, given a schedule, \(\psi = < SS_1, SS_2, \ldots, SS_i, \ldots,\)
4.4. Knowledge Structures and Methods

Time Intervals

\[ \lambda_1 \quad \lambda_2 \quad \lambda_3 \quad \lambda_4 \quad \lambda_{n-1} \quad \lambda_n \]

Figure 4.8: Λ Constraints on Scheduling.

\( SS_n >, \) if \( \psi \) satisfies all the problem constraints and \( \forall i: 1 \ldots n, SS_i \subset \text{Plan-Objs} \), then \( < SS_n, SS_{n-1}, \ldots, SS_i, \ldots, SS_1 > \) is the corresponding well-formed plan.

It is mandatory that in schedules, the plan objects are well-supported i.e., the support constraints for all the plan objects have to be satisfied. The Λ structures intrinsically capture these support relations and hence, scheduled with respect to the support constraints. The task of the scheduler is to configure the constituent structures given in a Λ into a single structure of a pre-defined number of temporal intervals and respecting the availability and other temporal constraints, such that the support constraints on the objects are preserved and the constraints active on each interval are satisfied. The properties of Λ structures and the availability constraints considerably reduce the complexity of the scheduling process in the CDSP problems.

Figure 4.8 summarises the useful properties of the Λ structure. Note that the objects in each of the constituent Λ structures may be scheduled in any interval against the time axis provided the Λ-support relations are preserved.

Search for a schedule is essentially a constraint satisfaction problem in which an input structure of plan objects \( CS \) is assigned to a specified set of temporal intervals such that the support relations between the plan objects are retained and all the \( \text{Sched-Cons} \): a combined set of plan object availability, \( \text{sched-pos-cons} \), user-specified precedent and coincident constraints, \( \text{plan-time-cons} \) and load constraints, are satisfied.

\[ ^{11} \text{A } \text{sched-pos-cons} \text{ constraint specifies the temporal intervals in which certain plan objects are to be placed.} \]
It requires creation of a schedule template of a sequence of sets (like the one given in Figure 4.9) for the non-committed temporal intervals specified. If there is not an executed plan then a template for all the intervals, as specified in the problem statement, is created. The search space is due to the availability constraints. These constraints specify for each plan object, the schedule interval(s) in which it can be assigned\(^\text{12}\). Possibly, there may be more than one interval for each object, and hence, the plan objects in CS may have multiple intervals in which they can be scheduled. This gives rise to multiple schedule possibilities. The \(\Lambda\) structures succinctly capture the support properties and facilitate search for a schedule. They are easily derived from the input CS structures.

Below, we illustrate the organisation of search for a schedule in terms of the \(\Lambda_1\) sequence from Figure 4.7, \(\Lambda_1 = \langle \{O_1, O_2\}, \{O_3, O_4\}, \{C\} \rangle, \langle \{O_5\} \rangle\), and the assumptions that the objects are available in the following intervals\(^\text{13}\):

- \(O_1\): \(\{I_3\}\)
- \(O_2\): \(\{I_3\}\)
- \(O_3\): \(\{I_2, I_3\}\)
- \(O_4\): \(\{I_1, I_2\}\), and
- \(O_5\): \(\{I_2\}\)

\(\Lambda_1\) contains \(\lambda_1 = \langle \{O_1, O_2\}, \{O_3, O_4\}, \{C\} \rangle, \lambda_2 = \langle \{O_5\} \rangle\).

In scheduling, after a \(\Lambda\) is derived from a CSS, each \(\lambda_i\) from the \(\Lambda\) is multiplied to get all the sequence possibilities for object assignments to intervals.

For the above example,

\[
\lambda_{11} = \langle \{(O_1, I_3), \{O_2, I_3\}\}, \{(O_3, I_2), \{O_4, I_1\}\}, \{C\} \rangle \\
\lambda_{12} = \langle \{(O_1, I_3), \{O_2, I_3\}\}, \{(O_3, I_2), \{O_4, I_2\}\}, \{C\} \rangle \\
\lambda_{13} = \langle \{(O_1, I_3), \{O_2, I_3\}\}, \{(O_3, I_3), \{O_4, I_1\}\}, \{C\} \rangle \\
\lambda_{14} = \langle \{(O_1, I_3), \{O_2, I_3\}\}, \{(O_3, I_3), \{O_4, I_2\}\}, \{C\} \rangle \\
\lambda_{21} = \langle \{(O_5, I_3)\} \rangle.
\]

Each of these sets of sequences are denoted by \(\lambda_{ti}\) where \(i\) is the index of the corresponding \(\lambda\).

In a pre-processing step to running the search, the candidate sequences from each set \(\lambda_{ti}\) is checked against each of the categories of constraints in the following order: ordering on temporal intervals, \((TI, <)\); any user-specified \textit{sched-pos-cons} constraints (given as \textit{TAKE-IN}\( <\text{temp-int}>\)); any user-specified coincident and precedent constraints, and filtered if the structures are inconsistent with these constraints. If any

\(^{12}\)The schedule intervals with a temporal ordering on them, \((TI, <)\), are defined in the CL language for each domain.

\(^{13}\)Note that the availability constraints are defined only for the plan objects.
of the sets become empty as a result of filtering then $\Lambda_1$ is unschedulable and the scheduling process on $\Lambda_1$ terminates with failure.

From the above example, $\lambda_{13}$ and $\lambda_{14}$ are inconsistent with interval ordering. In the former, $O_2$ supports $O_3$ but they are assigned to the same interval. Likewise, in the latter. Hence, the resulting set of structures is:

$\Lambda_{t1} = \{<\{(O_1, I_3), (O_2, I_3)\}, \{(O_3, I_2), (O_4, I_1)\}, \{C\}>\}$

The pre-processing behavior is achieved by the following function:

$\text{get-cand-struct}: \Lambda \times P \text{ CONST} \rightarrow P \lambda t$

$\text{get-cand-struct}$ takes a $\Lambda$ structure and a set of availability, any $\text{sched-pos-cons}$ and any user-specified precedent and coincident constraints and returns a set of well-formed temporal structures, if any. If no such structure can be found, then nil is returned.

The resulting sets of structures (called $\Lambda t$ structures) from the pre-processing step are well-formed with respect to plan object support, plan object availability, $\text{sched-pos-cons}$, and any user-specified coincident and precedent constraints. What is required now is a constraint satisfaction engine to perform a search for a combination of structures from each set of the $\Lambda t$ structures such that the load constraints and the plan time constraints are satisfied. The search has to be complete. Note, there may be constraints present in implicit form in the structures. This will result in a partially elaborated schedule.

In CSP formulation of scheduling of the well-formed temporal structures, we have $<X, \Lambda t, \text{Sched-Cons}>$, where each variable from a set of variables, $X = \{X_1, X_2, \ldots X_n\}$, has candidate sets defined by the domains:

$\Lambda_{t1} = \{\lambda_{t11}, \lambda_{t12}, \lambda_{t13}, \ldots\}$

$\Lambda_{t2} = \{\lambda_{t21}, \lambda_{t22}, \lambda_{t23}, \ldots\}$

$
\vdots$

$\Lambda_{tn} = \{\lambda_{tn1}, \lambda_{tn2}, \lambda_{tn3}, \ldots\}$.

The objective now is to find a combination of a set of variable assignments, for $(X_1, \ldots, X_n)$, from each of the respective domains ($\Lambda_{t1}, \ldots, \Lambda_{tn}$) and fill up a schedule template, like the one given in Figure 4.9, with this combination such that all the constraints in $\text{Sched-Cons}$ are satisfied.

The constraining property of $\Lambda s$ (see Figure 4.8) sanctions the use of value order-
strategy for the constraint satisfaction. Since, the At structure already gives a useful ordering between the structures (the earlier structures being more constraining), the variables, $X_1, X_2, \ldots, X_n$, are assigned values in order from $At_1, At_2, \ldots, At_n$.

The constraint satisfaction engine employs a state-based backtrack search to find each set of variable assignments at a time until a set, which results in a schedule, is found or all the possibilities are exhausted.

This constraint satisfaction engine is modeled by the function:

$\text{get-schedule}: P \lambda t \times P \text{CONST} \times P \text{TI} \rightarrow \text{Schedules} \cup \{\text{fail}\}$.

It takes a set of $\lambda t$ structures, a set of load and plan-time-cons constraints, and a set of unused temporal intervals and finds a schedule if one exists. $\text{fail}$ is returned if none of the structure combinations from $\lambda ts$ are schedulable.

In the search, the sets at each appropriate temporal level in the schedule are extended with the appropriate variable assignments and the resulting sets checked against the constraints from $\text{Sched-Cons}$ and any support constraints explicitly retained in the levels. For the former, according to the support property of the problems, it is imperative that the implicit prerequisite constraints appearing at a level $i$ in a well-formed temporal structure be satisfied by plan objects appearing in levels above $i$ in the temporal structure. If any of the support or $\text{Sched-Cons}$ constraints get violated then standard backtracking (see Chapter 5, page 116 for an algorithm) is invoked and other combinations are searched.

The chosen structures from the candidate sets are cumulatively combined to existing

---

14 A review of the CSP techniques is given in Chapter 2, page 19.
4.4. Knowledge Structures and Methods

Figures 4.10: Addition of a Well-Formed Temporal Structure to a Schedule.

schedules during the search. These combinations are explicitly retained at each state for later use in case of backtracking. The combination operation is captured by the following function:

\[ \text{add-struct: } \text{Schedules} \times \text{P } \lambda t \rightarrow \text{Schedules} \cup \{\text{fail}\}. \]

Addition of a \( \lambda t_i \) (an element of a \( \Lambda t \)), to a given schedule is illustrated in Figure 4.10. \( \psi \) and \( \psi' \) are of type \( \text{Schedules} \). We are given that \( C \) supports \( O_2 \) in \( \lambda t_i \). Hence, by the \( t u l \) property, it is placed in \( I_3 \).

Scheduling of units in degree planning is computationally inexpensive using the above problem solving framework because the number of intervals in each candidate set is usually small possible structure combinations is small. A unit is usually available in session 1, session 2 or over two sessions. Also, the levels of the units are given such as Comp3001 is a third year unit. The search is ordered by using the level information to choose (unit interval) tuples in which the interval matches with the level of the unit. Search progresses in the order of the temporal structure and if this fails to return a schedule then backtracking is invoked in order of the temporal structure: from the least constraining structure to the most constraining structure. Usually, the value ordering based on level-session combination makes search for a schedule tractable.
4.4.10 Addition of CSS to Existing Schedules

In the CDSP search for a schedule, given a state $S_i$ in the search, we may derive a schedule from the cumulative structure $S_i.CS$ extended by a conjunctive support structure, $css$ or it is possible to extend the existing schedule $S_i.schedule$ with $css$. We have presented the former approach in the above sections. The latter facility in the search minimises re-computations and hence, is more efficient.

In the current formulation of search as performed by the Model-Search problem solver, single partial schedules are retained at each state. Generation of a successor state involves extending the existing schedule with a $css$, if possible. If the schedule extension is not possible then for the desired completeness in search, a schedule is searched from $S_i.CS \oplus css$. In the IPLAN approach, multiple partial schedules are retained and there is a need to only extend the partial schedules by the $css$ increments.

In this section, we discuss how existing schedules can be extended. A conjunctive support structure, $css$, is added to an existing schedule, $S_i.schedule$, after $css$ is transformed into its corresponding well-formed temporal structure (i.e., a $At$ structure) as discussed in the previous sections. The resulting structure and the existing schedule are then combined using the $add-struct$ function such that all the scheduling constraints are satisfied. All the operations in the combination process is captured by the following function:

$extend-schedule: CSS \times Schedules \times P\ CONST \rightarrow Schedules \cup \{fail\}$.

$extend-schedule$ extends an existing schedule with a $css$ (of type $CSS$). It returns $fail$ if a consistent extension cannot found.

Tables F.1, F.2 and F.3 given in Appendix F contain the scheduling algorithms.

4.4.11 COBWEB Knowledge Clusters

Experiential knowledge in CDSP problems can be exploited in directing search in the problem solving. We use a clustering system called COBWEB developed by Fisher [57, 58] as a tool in deriving knowledge from previous cases. COBWEB is a computational model for classification of objects (input as instances) into concept hierarchies in which classification and learning proceed concurrently. It takes each object and sorts it down a concept hierarchy based on probabilistic and information-theoretic measures. A generalisation/specialisation hierarchy of clusters which reflect the degrees of similarities and differences between the instance attributes (also called features) results from the the classification. The algorithms and other features of the COBWEB system are described in detail in [57, 58] and reviewed in [52]. Here, we describe a
4.4. Knowledge Structures and Methods

typical input to the COBWEB system, and the form and the semantics of the output. We also explain how the structures produced by COBWEB are used in CDSP problem solving.

We have extended the COBWEB (we call it CDSP-COBWEB) implementation given in [52] to take a set of instances, perform the classification and returns a structure of clusters that is relevant to CDSP problem solving. It is modeled as a function:


For the purposes of classification, a plan is represented as a set of plan objects. A set of such sets represent a set of instances. This is input to CDSP-COBWEB. It returns a hierarchy of clusters. The nodes lower in the hierarchy are more general than the nodes higher up. An example on degree planning data is given in Appendix A. From this example, LStruct is:

Level 0: \{ all units \}

Level 1: \{ \}

Level 2: \{ GEOL2003 GEOL2004 GEOL3001 GEOL3004 \}

Level 2: \{ BOT1001 ZOOL1001 ZOOL2010 ZOOL2011 \}

Level 1: \{ MATH1011 MATH1012 MATH2001 MATH2002 MATH2003 MATH2011 MATH2012 MATH2013 MATH2014 MATH3001 MATH3005 STAT2001 STAT2002 STAT3003 STAT3009 \}

Level 2: \{ COMP1001 COMP1005 MATH1023 MATH2004 MATH3004 MATH3011 MATH3012 MATH3016 STAT1003 STAT1004 STAT2004 STAT3001 SWAA1001 \}

Level 1: \{ PSYC1001 PSYC2002 PSYC2007 PSYC2009 PSYC3001 PSYC3008 PSYC3010 STAT1003 \}

Level 2: \{ MATH1001 MATH1002 PSYC2001 PSYC2008 STAT1004 \}

Level 2: \{ STAT1004 STAT2001 STAT2002 \}

Level 1: \{ MATH2001 MATH2002 MATH2003 MATH2004 MATH2011 MATH2012 MATH2013 MATH3001 MATH3003 MATH3004 MATH3005 MATH3011 MATH3016 STAT3003 40084 40482 44048 \}

Level 2: \{ MATH3002 STAT2004 STAT3001 44349 \}

Level 2: \{ COMP2011 COMP2012 COMP2013 COMP3013 COMP3015 COMP3017 COMP3031 40385 44048 44349 \}

Level 2: \{ BIOC2001 BIOC2004 CHEM2005 CHEM2006 COMP1001 COMP1002 PHYS1001 \}

Level 1: \{ BIOC2001 BIOC2004 BIOC3002 BIOC3005 BIOC3007 BIOC3008 \}
Information contained in the clusters in this tree is used to focus selection of plan objects for satisfaction of implicit constraints. It provides an effective means for directing search. The plan objects present in an existing state of problem solving are used to find a matching cluster in LStruct and objects not already considered, are chosen from this cluster to carry the constraints to satisfaction. Advice on which plan objects to select, based on some existing problem solving state, may be generated for the user using the clusters—this feature is essential in an incremental and exploratory planning as we shall see in IPLAN in Chapter 7.

While the COBWEB clusters provide a good heuristic on selections of plan objects and advice generation, the clusters are based on previous cases in which all plan possibilities are not usually present. Also, since the formal constraints may change over time, some cases are bound to become invalid and hence the case database has to be regularly revised and COBWEB clustering done on each new instance of case database. Depending on the formal constraints existing at a time, the clusters may be different at different times.

The CDSP-COBWEB algorithm embeds the case database update mechanism. By this any cases which are inconsistent with an existing set of formal constraints get filtered prior to the classification. Typical changes include changes in

- availability of plan objects - whether or not, plan objects in the cases are available for future planning,
- plan object supports,
- plan object duration,
- plan object weight, and
- other formal constraints.
Chapter 5

Constraint Language: CL

5.1 Introduction

In this chapter, we present the language CL, the problem specification and input component of the CL-IPS problem solving language. Its syntax, and its semantics in terms of the state-space problem representation, the knowledge structures and a set of computational methods as abstracted from CDSP class of problems is presented. In the development of CL, the important concerns addressed are its richness in formulation of a wide range of CDSP problems and their input to the problem solver, and its support for user interaction. In particular, it is important that CL provides a communication medium between the user and the reasoning system such that user queries are easily communicated and knowledge from the user elicited from the user in a natural manner. There is a need for dynamic communication between the user and the system.

CL is a formal, precise, declarative language. It allows a descriptive statement of a CDSP problem in terms of constraints and other constructs in the language. The statements in CL are designed to be algorithmically interpreted and used in problem solving by the IPS component. CL allows the formulation of a wide range of CDSP problems i.e., CL is mappable on many CDSP problems and domains. It provides a medium for communication between the user and the IPS component, an essential component of CDSP problem solving.

Below, the syntax of CL is presented, and its semantics is given in terms of the state-space representation of the CDSP problems and the knowledge structures identified in the previous chapter. The syntax and semantics are formally defined so that problem statements can be algorithmically interpreted. The representational power of ORL is used for describing the objects, relations, and functions involved in the statement of a problem. All input data, including (both formal and informal) constraints, is
Chapter 5. Constraint Language: CL

encoded as well-formed sentences in CL prior to input. It is interpreted and internally represented as objects in ORL. The constraints are well-formed sentences in CL. CL also provides vocabulary for users to invoke an appropriate problem solving process.

5.2 CL Syntax

The domain definitions and the syntax of the CL language are given in Appendix B. Here, we summarise in a table (see Table 5.1) how different classes of constraints can be expressed in CL. From the table, the constraint classes 3 and 4 include the knapsack constraints, and the constraint classes 5-9 are temporal constraints.

The semantics for each of these constructs is given in the next section. In the semantic discussions, the following definitions for syntactic categories in terms of their corresponding primitive domains are used.

- plan-object \( \in \text{Obj-Id} \),
- plan-objects \( \subset \text{Plan-Objs} \) and denoted by a subset of \( \text{Obj-Id} \),
- deg-pref \( \in \text{Deg-Pref} \),
- pos-integer \( \in \mathbb{N} \),
- pos-real-number \( \in \mathbb{R}^+ \),
- class \( \subset \text{ATV} \),
- rel-op \( \in \text{Rel-Op} \),
- temporal-interval \( \in \text{Temp-Int} \), and

- temporal-class \( \in \text{Temp-Class} \)

5.3 CL Semantics

All the domain knowledge is input to the system through the CL language. CL allows the declarative formulation of CDSP problems. All the formal constraints are fixed and known \( a \text{ priori} \) to problem solving. The preferential constraints allow the user to dynamically communicate his/her preferences to the system.

In this section, we give the semantics of the constructs in CL.
### 5.3. CL Semantics

<table>
<thead>
<tr>
<th>Constraint Class</th>
<th>Expression in CL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Incompatibility</td>
<td>INCOMP <code>&lt;plan-object&gt;</code> <code>&lt;plan-objects&gt;</code></td>
</tr>
<tr>
<td></td>
<td>N-OF <code>&lt;pos-integer&gt;</code> <code>&lt;plan-objects&gt;</code></td>
</tr>
<tr>
<td>2. Exclusion</td>
<td>EXCLUDE <code>&lt;plan-objects&gt;</code></td>
</tr>
<tr>
<td></td>
<td>NOT-TAKE-OBJ-FROM <code>{&lt;obj-attrib.val&gt;}</code></td>
</tr>
<tr>
<td>3. Inclusion</td>
<td>INCLUDE <code>&lt;plan-object&gt;</code></td>
</tr>
<tr>
<td></td>
<td>SUM-WTS <code>&lt;class&gt;</code> <code>&lt;rel-op&gt;</code> <code>&lt;pos-real-number&gt;</code></td>
</tr>
<tr>
<td></td>
<td>N-OBJJS <code>&lt;class&gt;</code> <code>&lt;rel-op&gt;</code> <code>&lt;pos-integer&gt;</code></td>
</tr>
<tr>
<td></td>
<td>TAKE-OBJ-FROM <code>&lt;class&gt;</code></td>
</tr>
<tr>
<td>4. Load</td>
<td>SUM-WTS <code>&lt;temporal-interval&gt;</code> AT-MOST <code>&lt;pos-real-number&gt;</code></td>
</tr>
<tr>
<td>5. Plan-Time</td>
<td>PLAN-TIME <code>&lt;pos-integer&gt;</code> <code>&lt;temporal-class&gt;</code></td>
</tr>
<tr>
<td></td>
<td>PLAN-TIME-BETWEEN <code>&lt;pos-integer&gt;</code> <code>&lt;temporal-class&gt;</code> AND <code>&lt;pos-integer&gt;</code> <code>&lt;temporal-class&gt;</code></td>
</tr>
<tr>
<td>6. Sched-Pos-Cons</td>
<td>TAKE-IN <code>&lt;plan-objects&gt;</code> <code>&lt;temporal-interval&gt;</code></td>
</tr>
<tr>
<td>7. Precedent</td>
<td>TAKE-BEFORE <code>&lt;plan-object1&gt;</code> <code>&lt;plan-object2&gt;</code></td>
</tr>
<tr>
<td>8. Coincident</td>
<td>TAKE-CONC <code>&lt;plan-object1&gt;</code> <code>&lt;plan-object2&gt;</code></td>
</tr>
<tr>
<td>9. Availability</td>
<td>OBJ-AVAIL <code>&lt;plan-object&gt;</code> <code>{&lt;temporal-interval&gt;}</code></td>
</tr>
<tr>
<td>10. Change-Cons</td>
<td>SWAP <code>&lt;plan-object1&gt;</code> <code>&lt;plan-object2&gt;</code></td>
</tr>
<tr>
<td></td>
<td>KEEP-ALL-BUT <code>&lt;plan-objects&gt;</code></td>
</tr>
<tr>
<td></td>
<td>CHANGE-CATEG-PREF</td>
</tr>
<tr>
<td>11. Conditional</td>
<td>WHAT-IF <code>{constraints classes 2, 3, 5, 6, 7, 8 and 10 from above}</code></td>
</tr>
</tbody>
</table>

Table 5.1: A Summary of Constraint Expressions in CL.
5.3.1 Background Knowledge

The background knowledge is formalised, input to the system and represented as objects in ORL. The knowledge retrieval functions and integrity constraint checking functions are appropriately established for the objects. (The section on knowledge representation discusses the ORL representation issues, see §4.2 on page 53).

i). Global sets of objects are defined for the domain under consideration. Also, any formal relations between these sets are user-defined using the INC (for strict set inclusion) and INCL (for set inclusion) statements in CL. These definitions are global.

ii). \(<\text{temporal-interval-sequence}>\) is a sequence of temporal intervals in the basic time units as given in the set \(\text{Temp-Int}\). The ordering relation between the intervals is important.

The composite intervals, as given by the set \(\text{Temp-Class}\), are sequence combinations of the primitive intervals from \(\text{Temp-Int}\). The elements of \(\text{Temp-Class}\) are unique names for these combinations. \(<\text{named-temp-interval-combinations}>\) describes these sequence combinations.

iii). Plan objects are the objects that are planned. They are input, and internally represented as ORL objects as illustrated in Figure 4.3. The various attributes\(^1\) of a plan object are packaged in its object structure. Selection of attribute names and values are restricted to the predefined sets \(AT\) and \(ATv\) respectively. All the plan objects are uniquely identified by an identifier from the set \(\text{Obj-Id}\).

The attributes of these objects which are essential to the CDSP problems are the object weights (given as a positive real number, \(\mathbb{R}^+\)), object availability (a constraint specifying a set of temporal intervals from \(\text{Temp-Int}\) in which an object is available for assignment), and object supports (constraints which specify the precedence and co-requisite supports for the object).

iv). The formal constraints are input to the system as statements in CL, interpreted and captured as ORL objects of Figure 4.4. All the possibly expressible constraints in CL have a set of tester (or evaluator) predicates for testing their satisfaction and satisfier functions for finding ways of satisfying the constraints. (The

\(^1\)Note that ORL also allows any meta-information on objects to be packaged as slots in object representations.
semantics of these operators is dependent on the class the constraints belong to as explained later on page 90.) These functions are available in a repertoire of pre-defined functions from which the appropriate predicates and satisfier functions are inherited by the constraint objects.

v). The LStruct is an ORL structure of knowledge clusters derived from the experiential knowledge. The induction of these structures and their representation have been discussed in §4.4.11 and Appendix A. Their use in focusing search is an important feature of CDSP problem solving and presented in the latter sections. They are also used in advice generation in user-directed problem solving as modeled by IPLAN.

vi). Statistical information is drawn from previous successful plans given as experiential knowledge, and used as heuristics in the search process. Aggregate weight information on different classes of objects from each previously successful plan is a useful heuristic.

Other useful information which are derived from frequencies of occurrences are the popularity of given plans, popularity of plan objects, and popularity of plans with respect to some given set of attributes. For example, in degree planning, information on number of units taken from each department is useful. These pieces of information describe the Case-Info.

Case-Info objects have been described for the degree planning problem but it is not abstracted for a general CDSP problem. A general definition for Case-Info and method for its derivation are beyond the scope of this thesis.

vii). The Com-Cons is a set of executed plan objects (called XPLAN) uniquely identified by an identifier from Plan-Id. An XPLAN is represented as an object in ORL. A schema for an XPLAN object is given in Table 5.2. In the table, \( Id \) is the executed plan identifier, \( PEST \) is the plan execution start time (e.g., degree enrolment date), \( Results \) is a set of tuples: \( \{(pest \ (\{obj_1 \ res \ int\}) \ (obj_2 \ res \ int) \ ... \ (obj_k \ res \ int))\} \), where \( pest \) is the plan execution start time, and the tuple \( (obj \ res \ int) \) represents an executed plan object, result of its execution, and the temporal interval in which it was executed respectively. In degree planning, the results are the grades attained for each unit undertaken and the intervals are sessions in which units were taken. Other attribute-value pairs, as required, may be defined in the XPLAN schema. For example, in the degree planning domain, we need to package the name of the degree and the faculty to which it belongs.
5.3.2 Constraint Statements

In both Model-Search and IPLAN problem solvers, constraints are used in guiding the search process. The selection of constraints and their application in directing search, detection of inconsistencies in sets of constraints active on a search process and recovery from it, and the representation and management of "reasoned" knowledge are dependent on the control regime used. These functionalities belong to the IPS component and they are addressed in both, the Model-Search and IPLAN computational models, in the later chapters. Here, we present the semantics of each constraint statement, as given in CL, in terms of a Model-Search problem solving state, $S_i: \langle AC, C, EX, X, CS, PC, schedule \rangle$. (Note in a $CS$, plan objects from both executed and unexecuted plans are captured. The executed plan objects are retained as xplans.) The IPLAN problem solver uses different structures in problem solving but the semantics of application of constraints is the same as in Model-Search.

In the CDSP problem solving, the constraints are used both, as Boolean constraints, and as generators. In the former use, the plans are checked through the constraint tester predicates whether they satisfy the constraints, and in the latter use, information contained in the constraints are exploited in searching for plan objects and their supports to add to an existing plan such that it satisfies the constraint. The second semantics of constraints aid in organising and guiding search for consistent plans. This constraint-directed search is an extension to heuristic search paradigm\(^2\).

\(^2\)Problem representation as state-space search and the characterisation of a problem state are given on pages 60 and 61 respectively.

\(^3\)Note that a plan is a schedule in reverse order; see page 73.

\(^4\)Pearl, in [121], has presented various theories on the use of heuristics in intelligent search strategies.
Prior to giving the semantics of the constraints in CL, we give below additional notations and definitions that we use in the semantic discussions.

`CONST` is a set of all possible constraints defined for a problem. These constraints are sentences in CL. $\Delta$ is a set of such constraints; $\Delta \subseteq CONST$. A set of local constraints (LC) is identified from $\Delta$ and $AC$ in a given state. These are the equality, non-load-upper-limit, exclusion, incompatibility, load (which are upper-limit constraints) and temporal constraints. They are retrieved from some state $S_i$ through $(\text{get-local-cons } S_i, AC \Delta)$. These constraints are used in local search i.e., in each successor state generation. For their effective and efficient use in different stages of local search, LC is further divided into local selection constraints (LSC): equality, non-load-upper-limit, exclusion and incompatibility constraints; and Sched-Cons (as defined on page 73): load and all the temporal constraints. A $S_i.CS$ is locally-consistent iff it is consistent with all the local selection constraints and is schedulable.

A consequence of a constraint $C$ is a mapping from $C$ to a set of plan possibilities, each element of which satisfies $C$. Satisfaction (or processing) of a constraint $C$ is a mapping of a constraint to its consequence. It requires search for plans which satisfy $C$. According to the Compactness Theorem [20], a set of constraints has a model if and only if every finite subset of the set has a model. Hence, if some $\Delta'$ is a subset of $\Delta$ and does not have a model (i.e., $\Delta'$ is not consistent) then $\Delta$ does not have a model. Testing a constraint $C$ on a state $S_i$ means to check whether or not $S_i.schedule$ satisfies $C$. Application of a constraint $C$ to a state $S_i$ means to generate another state $S_{i+1}$, if possible, using $C$ and information contained in $S_i$ such that $C$ is satisfied in $S_{i+1}$. In each state, $S_i$, a constraint is processed (or applied) if it is in the sequence given by $S_i.PC$ or in the set of local constraints defined in $S_i$. It is unapplied otherwise.

There are various different kinds of constraints which can be expressed in CL. Their classification into inclusion, exclusion, support, incompatibility, temporal, conditional and case-based constraint sets is one natural and useful classification. We give the constraint semantics based on this classification. However, different semantics of constraints, even in each such classification, and partial orderings between them permit their use in facilitating their automatic selections and applications in different stages of the search process. (This is addressed in Chapter 6.)

The case analysis of semantics for each of the constraint categories is given below. We give the tester (or evaluator) and satisfier functions for each category of constraints. The tester functions take a state and a constraint and determine whether the constraint is satisfied, satisfiable, unsatisfiable with respect to the state. It returns
true if the constraint is satisfied, false if the constraint is violated (hence unsatisfiable), and satisfiable (i.e., undetermined) if the constraint is not satisfied and not violated. Hence, the co-domain of the tester functions is: Cons-Sat = Boolean ∪ {satisfiable}. These satisfiability notions are made clear in the following semantic discussions for each category of constraints.

Note that, in all the function definitions used in the following discussions, datatypes are specified using ‘:’. For example, declaration of a natural number n is given by n:N. Also, throughout this thesis, Lisp syntax is used in function calls.

5.3.2.1 Incompatibility Constraints

There may be plan objects which cannot appear together in the same plan. These restrictions are specified by the incompatibility constraints. There are typically two groups of incompatibility constraints.

i). One group is declared in CL as an attribute to a plan object in the syntax:

\[ \text{INCOMP}<\text{plan-object}>>\text{plan-objects}. \]

(This is referred to as Type 1 incompatibility.) This constraint specifies that if some given plan object, plan-object is present in some set (say, PO) then all the elements of the set plan-objects cannot be in PO. If this is so, then PO is inconsistent with the constraint.

Tester function:

\[ \text{incomp-tester}: P \text{ Plan-Objs} \times \text{CONST} \rightarrow \text{Cons-Sat} \]

Input: PO ⊆ Plan-Objs, C:CONST

\[ \text{if } C.:\text{plan-objects} \cup \{ C.:\text{plan-object}\} \subseteq PO \text{ then (return false)} \]

else (return true) fi

ii). The second group is given in the CL syntax:

\[ \text{N-OF } <\text{number1 }>>\text{plan-objects}. \]

(This is referred to as Type 2 incompatibility.)

Here, the constraint specifies restriction on the number of plan objects from certain plan object sets which can occur together in the same set (say, PO). If more than number1 number of plan objects from plan-objects appear in PO then PO is declared inconsistent with the constraint.

Tester function:
5.3. CL Semantics

incompat-test: $P \times \text{CONST} \rightarrow \text{Cons-Sat}$

Input: $PO \subseteq \text{Plan-Objs, } C: \text{CONST}$

\[
\text{if } (\text{count-objs (intersect } PO \text{.plan-objects)}) > C.\text{number1 } \text{then} \\
\text{(return false)} \\
\text{else (return true)}
\]

\[\text{fi}\]

The following function performs the combined incompatibility evaluation on a set of plan objects for some given set of incompatibility constraints.

incomp-inconsistent?: $P \times \text{P CONST} \rightarrow \text{Cons-Sat}$

Input: $PO \subseteq \text{Plan-Objs, } IC$ as a set of incompatibility constraints ($IC \subseteq \text{CONST}$).

\[\text{until } IC = \phi \text{ do} \]

\[ic \leftarrow (\text{choose } IC) \]

\[IC \leftarrow IC \setminus ic \]

\[\text{if } (\text{type1p } ic) \land ic.\text{plan-objects} \cup \{ic.\text{plan-object}\} \subseteq PO \text{ then} \]

\[\text{(return false)} \]

\[\text{elseif (count-objs (intersect } PO \text{.ic.plan-objects)}) > ic.\text{number1 } \text{then} \]

\[\text{(return false)} \]

\[\text{fi}\]

\[\text{enddo}\]

\[\text{(return true)}\]

5.3.2.2 Inclusion Constraints

The inclusion constraints specify, both extensionally and intensionally, the plan objects and implicitly, their support structures (captured as conjunctive support structures), which have to be included in plans for these constraints to be satisfied\(^5\). Their satisfier operators perform plan object selections and effect state expansions.

The inclusion constraint categories in CL are given as \text{<inclusion>}, \text{<weight-requirements>}, and \text{<load-constraints>}. It is useful to further divide constraints from

\(^5\)Application of the inclusion constraints result in commitment of new objects to some existing plan. They are called the generator constraints because of this additive effect on their application to some existing state.
these categories into upper-limit, lower-limit, equality and non-limit constraint sets. The former three are knapsack partitions and the latter is a set of ground constraints of the kind INCLUDE $<\text{plan-object}>[<\text{deg-pref}>]$. In search, it is favorable to apply the non-limit, and then the lower-limit and equality constraints in order from most constraining to least constraining and also respecting the upper-limit and other local constraints. Then, the upper-limit constraints (such as the load constraints) can be applied as generators. There may also be the case that some lower-limit constraints are weaker than some upper-limit constraints. In this case, the stronger upper-limit constraints are applied until the unapplied lower-limit constraints become stronger and their domains become computable—domains of constraints are sets of sets of plan objects dynamically computed during. The dynamic creation of domains for inclusion constraints, and the ordering and selection of inclusion constraints for application in search have been deferred until Chapter 6.

Application of an inclusion constraint $C$ to some state $S_i$ results in another state $S_{i+1}$ if $C$ is satisfiable with respect to $S_i$. It involves checking whether $S_i.CS$ satisfies $C$. If $C$ is not satisfied then a conjunctive support structure $CSS_x$ is searched which, when added to $S_i.CS$ satisfies $C$. The aggregation $S_i.CS \oplus CSS_x$ has to respect the unique support property of plan objects in plans, and it has to be schedulable and locally consistent. If such a structure can be found then $S_{i+1}$ results on which $C$ evaluates to true, and $S_i.AC \cup \Delta$ is declared satisfiable. Otherwise, $S_i.CS \oplus CSS_x$ is declared untenable (or nogood) with respect to $C$.

The inclusion constraints have the tester and satisfier operators, and the scheduling operators all of which combine to form transition operators in search. If a successor state $S_{i+1}$ cannot be generated through a constraint $C$ then $C$ is declared unsatisfiable with respect to $S_i$. Note this results in a dead-end situation in the search path. Still there may be other unexplored paths which may lead to solutions. Hence, at a dead-end $S_i, S_i.AC \cup \Delta$ is not inconsistent because other choices of values for variables may lead to a solution. It is inconsistent only if all possibilities are exhausted.

The inclusion constraint categories identified above differ in their satisfiability semantics in the following ways. Note, in search, it is assumed that an existing set of constraints $S_i.AC \cup \Delta$ is satisfiable unless at least one of the constraints in constraint sets in the succeeding states, $S_{i+1}.AC \cup \Delta \ldots S_n.AC \cup \Delta$, become unsatisfiable. Once a constraint $C$ from $S_i.AC \cup \Delta$ becomes unsatisfiable, $S_i.CS$ is declared untenable with respect to $C$. 
Non-Limit Constraints

A non-limit constraint $C$ is satisfied in $S_i$ iff the plan objects given in the set $C$.plan-objects are contained in $S_i.CS$. It is satisfiable with respect to a $S_i$ iff the $S_i.CS$ can be extended to accommodate all the plan objects in $C$.plan-objects without violating any other constraint. To satisfy these constraints means to extend $CS$ to include those plan objects specified in the constraint but currently missing from $CS$. In an empty plan, these constraints are satisfiable.

Equality Constraints

An equality constraint $C$ is satisfied in a state $S_i$ iff $S_i.CS$ contains a set of plan objects which meet certain given attribute restrictions as specified in $C$ and has the aggregate weight (or count) of the plan objects in the set equal to $C$.number. If the aggregate weight is less than $C$.number then $C$ is satisfiable by extending $S_i.CS$ and $C$ can be used as a generator constraint. If it is greater than $C$.number then $C$ is unsatisfiable (or inconsistent) with respect to $S_i.CS$. To satisfy the satisfiable constraints means to extend the existing $CS$ to include plan objects from classes specified by the constraints such that the equality condition is satisfied. In an empty plan, these constraints are satisfiable.

Lower-Limit Constraints

A lower limit constraint $C$ is satisfied in a state $S_i$ iff $S_i.CS$ contains a set of plan objects which meet certain given attribute restrictions as specified in $C$ and has the aggregate weight (or count) of the plan objects in the set greater than or equal to $C$.number. It is satisfiable iff an extension of $S_i.CS$ results in the satisfaction of $C$ without violating any other constraint, and unsatisfiable (or inconsistent) with respect to $S_i$ otherwise. To satisfy these constraints means to include plan objects from classes specified by constraints such that the lower-limit condition is satisfied. In an empty plan, these constraints are satisfiable.

Upper-Limit Constraints

An upper limit constraint $C$ is satisfied in some state $S_i$ iff $S_i.CS$ contains a set of plan objects which meet certain given attribute restrictions as specified in $C$ and has the aggregate weight (or count) of the plan objects in the set less than or equal to

\footnote{For each category of inclusion constraints, we adopt different meanings to satisfy when it is used in the rest of the thesis.}
Chapter 5. Constraint Language: CL

C.number. C can be used as a generator constraint if C is satisfied and the aggregate weight is lower than the limit. If the aggregate weight is greater than C.number then C is unsatisfiable (or inconsistent) with respect to S_i. To satisfy these constraints means to include plan objects from classes specified by constraints such that the aggregate weight is equal to the upper-limit weight specified in the constraint. In an empty plan, these constraints are satisfied.

The search for a conjunctive support structure to satisfy C involves finding an expansion for C with respect to the existing knowledge in S_i. An expansion for C is an element of C's expansion-set. An expansion-set of an inclusion constraint C is a set of subsets of Plan-Objs with the following relationships with each kind of constraint and some existing state S_i.

i). If C is a non-limit constraint then the expansion-set of C contains one element which is C.plan-objects \ (get-plan-objs S_i.CS).

ii). If C is a lower-limit constraint then the expansion-set of C is the set of all subsets of Plan-Objs, each element of which when combined with S_i.CS will satisfy C. In using C as a generator constraint, the conjunctive support structure for an element from the expansion-set which, when combined with S_i.CS results in an aggregate weight (or count) of plan objects in the combination equal to or minimally exceeding C.number is chosen.

iii). If C is an upper-limit constraint then the expansion-set of C is the set of all subsets of Plan-Objs, each element of which, when combined with S_i.CS will satisfy C. An element from the expansion-set of C is found which, when combined with S_i.CS results in an aggregate weight (or count) of plan objects in the combination equal to or minimally less then C.number.

iv). If C is an equality constraint then the expansion-set of C is the set of all subsets of Plan-Objs, each element of which when combined with S_i.CS will satisfy C. In using C as a generator constraint, the conjunctive support structure for an element from the expansion-set which, when combined with S_i.CS results in an aggregate weight (or count) of plan objects in the combination equal to C.number is chosen.

The expansion-sets in (ii), (iii) and (iv) are combinatorial and may not be easily computable. The consequent set information, and the LStruct clusters are exploited in guiding selections of plan objects from these sets. The constraint selection algorithm
selects constraints in order from most constraining to least constraining and helps fur-
ther in focusing search.

In search, first an inclusion constraint $C$ is chosen for application. Then an expansion
is chosen for $C$ and a conjunctive support structure\(^7\) for the expansion is searched.
The chosen conjunctive support structure $CSS_x$, when combined with $S_i.CS$ has to
have unique object supports, be consistent with the local constraints and be schedu-
lable. If such a conjunctive support structure is found then a new state is generated,
otherwise $S_i.CS \oplus CSS_x$ is recorded in a nogoods database, another expansion for $C$
is chosen, and the process repeated until a $CSS_x$ is found such that $S_i.CS \oplus CSS_x$ is
consistent or all possibilities are exhausted. If such a $CSS$ cannot be found then no
extension of $S_i.CS$ would lead to a solution and standard backtracking is invoked.

In general, given a nogoods database\(^8\) of sets of plan objects, a state $S_i$ and an
xplan, the application of an inclusion constraint $C$ to a state $S_i$ can be summarised in
the following steps.

1. Check $S_i.CS$ for satisfaction against $C$ and take appropriate action:

   \begin{enumerate}
   \item case $C$: equality
   \begin{enumerate}
   \item if (satisfies? $C S_i.CS$) then
   \begin{enumerate}
   \item $S_{i+1} \leftarrow S_i$
   \item (push $C S_{i+1}.PC$)
   \item (return true)
   \end{enumerate}
   \item elseif (satisfiable? $C S_i.CS$) then continue
   \item else (return fail)
   \end{enumerate}
   \item case $C$: upper-limit
   \begin{enumerate}
   \item if (satisfies? $C S_i.CS$) then
   \begin{enumerate}
   \item $S_{i+1} \leftarrow S_i$
   \item (push $C S_{i+1}.PC$)
   \item else (return fail)
   \end{enumerate}
   \item case $C$: lower-limit
   \begin{enumerate}
   \item if (satisfies? $C S_i.CS$) then
   \begin{enumerate}
   \item $S_{i+1} \leftarrow S_i$
   \item (push $C S_{i+1}.PC$)
   \item (return true)
   \end{enumerate}
   \end{enumerate}
   \end{enumerate}
\end{enumerate}

\(^7\)See page 64 for the definition of conjunctive support structures.
\(^8\)nogoods database is discussed in Chapter 6, see page 134.
elseif (satisfiable? C S_i.CS) then continue
else (return fail)

2. Find the difference, $\delta_C$, with respect to $C$ between the $CS$ in the current state $S_i$ and $CS$ in the goal state $S_g$. Note, this difference is not between the two states as employed in best-first search approaches and realised in systems such as STRIPS [55] (as means-ends analysis). Rather, it is a measure of how close a particular inclusion constraint is to satisfaction. The state difference calculations is irrelevant to CDSP problem solving as the goal state is not known a priori. New constraints emerge during search and in the resulting goal state all the constraints have to evaluate to true.

3. Use $\delta_C$ to find the most “promising” set of plan objects which is an expansion of $C$, $EX$, with respect to $S_i.CS$, which, when added to $S_i.CS$ would satisfy $C$. This set is searched in the following way.

4. Retrieve from the knowledge base, the consequent set ($Conseq-Set$) for all plan objects in $S_i.CS$. The $Conseq-Set$ of a set of plan objects ($PO$) is recursively defined as a set of those plan objects which have at least one object in $PO$ as their support and their consequent sets. In problem solving, each found consequent set is cached out as a set of tuples and globally maintained avoiding their expensive re-computations. For example, in the degree planning problem, the set \{(Math1001 \{Math2001, Math2002, Math2003, Math2004\}), (Math2002 \{Math3005, Math2002, Math2004\}), (Math2003 \{Math2004, Math3003\}), (Math2004 \{Math3005\}), (Math3005 \{Math3018\})\} is generated as the consequent set for Math1001 and recursively, its consequents.

5. Use the clustered knowledge base $LStruct$ (as found by COBWEB on case data), if available, the $Conseq-Set$ and the bias sequence from the category-preferences to find an expansion of $C$ i.e., a selection of plan objects to satisfy $C$ in the new state $S_{i+1}$.

6. Find the set of conjunctive support structures $CNSS$ for the plan object selection $EX$.

7. Filter those structures in $CNSS$ which are common to $S_i.CS$ and have different supports—according to the unique support property of CDSP problems, only one conjunctive support is sufficient for a plan object’s existence in a plan.
8. Check the consistency of each extension \((CS')\) of \(S_i.CS\) with each \(css\) from \(CNSS\) with each of the constraints from \(LSC\). If an extension is inconsistent then record the \(css\), and add the set of plan objects contained in \(CS'\) to the nogoods database along with a set of labels of the constraints that are violated. Filter all the recorded \(css\) structures which result in inconsistencies from \(CNSS\).

9. If \(CNSS = {}\) and \(expansion-set \neq {}\) then make next selection for expansion, go to 6. Otherwise if \(CNSS = {}\) then return \(\text{fail}\).

10. Put the structures in \(CNSS\) in a sequence, from maximally to minimally matching of structures with \(S_i.CS\). The matching is determined by counting the number of common nodes in each structure.

11. Find a \(CSS, CSS_x\), from the sequence, such that \(CSS_x \oplus S_i.CS\) is schedulable. This requires finding a schedule for \(CSS_x \oplus S_i.CS\) which satisfies all the load and temporal constraints given in \(Sched-Cons\).

As an efficiency measure, the existing schedule is first extended with a \(CSS_x\), and if a consistent extension is not possible then an exhaustive search for a schedule on \(S_i.CS \oplus CSS_x\) is performed.

12. If \(S_i.CS \oplus CSS_x\) is inconsistent and \(expansion-set \neq {}\) then record the set of plan objects in \(S_i.CS \oplus CSS_x\) as a nogood, make next selection for expansion, go to 6,

13. elseif \(S_i.CS \oplus CSS_x\) is inconsistent and \(expansion-set = {}\) then return \(\text{fail}\),

14. else generate and return \(S_{i+1}\).

The various pieces of heuristic information that are exploited in guiding selection of plan objects and and their conjunctive support structures when applying the inclusion constraints are:

- unique support property of plan objects to avoid inclusion of multiple conjunctive supports for same plan objects—it is required that this property is met by all selections made,

- bias in plan object selections as a result of user-specified category preferences,

- consequent sets of plan objects present in the existing state,

- COBWEB knowledge clusters, and
• the conjunctive support structure is selected from a set of possibilities, which maximally overlaps with the existing cumulative structure.

Since, the goal state is not known until the problem solver finds one, heuristic evaluation functions cannot be used and hence, existing best-first approaches (as reviewed in [121]) are not directly applicable.

As part of partial evaluation, an extension is found which does not violate any of the upper-limit and equality constraints.

**INCLUDE <plan-object> [<deg-pref>]**

This constraint specifies that the plan objects declared in the plan-object set are to be present in the plan being searched for the constraint to be satisfied. The optional degree of preference information is used in the search process as a heuristic in guiding the selection of inclusion constraints for application. The order of selection is from most to least preferred.

The tester function for the constraint is:

\[test-obj-inclusion: S \times CONST \rightarrow Cons-Sat\]

Input: \(S; C\) is an instance of the above constraint \((C;CONST)\)

\[
\text{if } C.\text{plan-objects} \subseteq (\text{get-plan-objs } S; CS) \text{ then (return true),}
\]

\[
\text{else (return satisfiable) fi}
\]

The satisfier function takes a state and an instance of the above constraint and finds a new state, if possible, with the constraint satisfied in it. If such a state cannot be found, \text{fail} is returned. This behavior is modeled by the following function.

\[process-obj-inclusion: S \times CONST \rightarrow S \cup \{\text{fail}\}\]

Input: \(S; C\) as an instance of the above constraint, \(C;CONST\)

\[
\text{if } (test-obj-inclusion } S; C) = \text{true then}
\]

\[
\text{(return (update } S; C))
\]

\[
(*C \text{ is already satisfied.} *)
\]

\[
\text{else}
\]

\[
(*C \text{ is not satisfied in } S_i. A } S_{i+1} \text{ is searched such that } C \text{ is satisfied in it.} *)
\]

\[9\] Relevance of \(A^*\) search to and its potential use in CDSP problem solving are assessed in Chapter 6.

\[10\] The use of preferential information helps in guiding search at the choice points. This is discussed in the search strategies for the CDSP class of problems in the later chapters.
5.3. CL Semantics

a-set ← C.plan-objects \ (get-plan-objs S_i.CS)
(*Gets a set of possible objects to choose from.*)

css-cand ← (local-select (dof (get-pi a-set)) S_i.CS)
(*css-cand is a sequence of structures, each of which would lead to the satisfaction of C.*)

new-sched ← (find-schedule css-cand S_i)
if new-sched ≠ nil then S_{i+1}.EX ← a-set
(return S_{i+1})
else (return fail)
(*C is unsatisfiable.*)
fi
fi

local-select finds a sequence of those CSS structures each of which, when added to an existing CS results in a CS that is consistent with the existing LSC constraints. First, each of the extensions are checked with the nogoods database and the extensions which contain objects that are known to lead to failure are filtered.

local-select: P CSS × CSS → CSS-SEQ
Input: S_i.CS:CSS, css:Set-Of CSS
Output: A sequence of CSS or null (an empty sequence)
(*The css structures which result in untenable extensions of a given CS are filtered and the remaining structures are ordered based on their degree of matching with the existing CS, S_i.CS.*)

for each cs1 in css

CS' ← cs1 ⊕ S_i.CS
if (in-nogoods? (get-plan-objs CS')) then nogoods ← nogoods ∪ {cs1}
fi
(*in-nogoods? finds out whether a set of plan objects has a subset in the nogoods database.*)
endfor

css ← css \ nogoods

css1 ← {}
until \( css = \phi \) do

\[
\text{cand} \leftarrow (\text{choose } css) \\
\text{css} \leftarrow \text{css} \setminus \text{cand} \\
\text{if} \ (\text{not} \ (\text{incompat? } S_i.CS \oplus \text{cand})) \ \text{then} \\
\text{cssl} \leftarrow \{\text{cand}\} \cup \text{cssl} \ \text{fi} \\
\text{(*Conjunctive support structures which result in structures containing in-} \\
\text{compatible plan objects are filtered. Note, the Type 1 incompatibilities are} \\
cached in the nogoods database and all the Type 2 incompatibilities are only} \\
\text{checked by the function incompat?.*)} \\
\text{endifdo} \\
\text{if } \text{cssl} = \{\} \ \text{then} \ (\text{return } \text{null}) \\
\text{else} \ (\text{return} (\text{order-max-match } S_i.CS \ (\text{filter-limit-violations (get-upper-limit-equality-} \\
\text{cons (}\Delta \cup S_i.AC)) \ (\text{filter-multiple-support } S_i.CS \ \text{cssl})))) \\
\text{fi} \\
\]

\textit{filter-multiple-support: } CSS \times P \ CSS \rightarrow P \ CSS \\
\text{Input: } T1:CSS, \text{css } \subset CSS \\
\text{for each } T2 \text{ in css} \\
\text{diff} \leftarrow (\text{intersect } T1 \ T2) \\
\text{if } \text{diff} = \phi \ \text{then} \ (\text{return} \ \{T2\}) \\
\text{elseif} \ (\text{check-unique-supports } T1 \ T2) \ \text{then} \ \text{match-set} \leftarrow \text{match-set} \cup \{T2\} \\
\text{(*Check whether all arcs in } T1 \text{ are also present in } T2 \text{ as required by the} \\
\text{unique support property of plan objects.*)} \\
\text{fi} \\
\text{endfor} \\
\text{endfor} \\
\text{enddo} \\
\text{if } \text{cssl} = \{\} \ \text{then} \ (\text{return } \text{null}) \\
\text{else} \ (\text{return} (\text{order-max-match } S_i.CS \ (\text{filter-limit-violations (get-upper-limit-equality-} \\
\text{cons (}\Delta \cup S_i.AC)) \ (\text{filter-multiple-support } S_i.CS \ \text{cssl})))) \\
\text{fi} \\
\]

\textit{filter-multiple-support} filters all those conjunctive support structures from \text{css} which 
\text{are also in } S_i.CS \text{ and which differ in supports in the two structures. The support prop-} 
\text{erty of the problem dictates that in a plan there has to be a single conjunctive support}
structure for a plan object. This function filters any occurrence of multiple supports for same objects violating the unique support property of conjunctive support structures CSS and cumulative structures CS. filter-limit-violations filters those conjunctive support structures which, when added to an existing $S_i.CS$ results in a $CS$ which violates the equality or upper limit constraints—a lookahead step in the partial evaluation. order-max-match matches each of a given set of conjunctive support structures with some given $S_i.CS$, and orders them based on degree of their similarities by matching the number of same objects. It returns a sequence of these structures ordered from maximally to minimally similar.

**update-state:** $S \times CONST \rightarrow S$

Input: $S_i:S$ and $C:CONST$

$$S_{i+1} \leftarrow S_i$$

$$S_i.C \leftarrow C$$

(push $C S_{i+1}.PC$)

(*Update the sequence of satisfied constraints.*)

(return $S_{i+1}$)

For each structure in the sequence produced by local-select, find-schedule finds a schedule, if possible, and an updated state as a side-effect, from

i). an existing schedule and a set of CSS structures, and failing this,

ii). extensions on an existing cumulative CS structure.

It returns nil if it fails to find a schedule which is consistent with a set of existing local constraints.

**find-schedule:** $CSS-SEQ \times S \rightarrow Schedules \cup \{fail\}$

Input: css-cand:CSS-SEQ, $S_i:S$

**until** css-cand = null do

$$css1 \leftarrow \text{pop css-cand}$$

$$up-LC \leftarrow \text{get-local-cons} S_i.LC css1$$

$$schl \leftarrow \text{get-extension} css1 up-LC S_i$$

if schl $\neq$ false then

(*If a consistent schedule is found then generate the successor state, $S_{i+1}$, and update the information in it.*)
Chapter 5. Constraint Language: CL

\[ S_{i+1}.AC \leftarrow S_i.AC \cup (\text{get-cons cssl}) \]
\[ S_{i}.C \leftarrow C \]
\[ S_{i+1}.X \leftarrow \text{cssl} \]
\[ S_{i+1}.CS \leftarrow S_i.CS \oplus \text{cssl} \]
\[ S_{i+1}.PC \leftarrow (\text{add } C \quad S_{i}.PC) \]
\[ S_{i+1}.schedule \leftarrow \text{sch1} \]
\[ (\text{return sch1}) \]

\textbf{Enddo}

\textbf{else} (\text{return fail})

(*If a schedule cannot be found from the exhaustive search then fail is returned.*)

\textbf{get-extension}: \text{CSS} \times P \text{CONST} \times S \rightarrow \text{Schedules} \cup \text{Boolean}

\textbf{Input}: css:CSS, LC \subseteq \text{CONST}

\textbf{if} (\text{incomp-inconsistent?} \quad (\text{get-plan-objs ess}) \quad \text{LC}) \textbf{then} (\text{return false}) \textbf{fi}

\text{sch1} \leftarrow (\text{extend-schedule css } S_{i}.schedule \text{ LC})

(*Search for an extension of \text{S}_i.\text{schedule} with a CSS.*)

\textbf{if} \text{sch1} \neq nil \textbf{then} (\text{return sch1})

\textbf{else}

(*If an extension of \text{S}_i.\text{schedule} with a CSS cannot be found then search for a schedule from \text{S}_i.\text{CS} and a set of CSS.*)

\text{css1} \leftarrow S_{i}.CS \oplus \text{css}

\text{sch1} \leftarrow (\text{schedule css1 LC} \quad (\text{get-unused-intervals xplan}))

\textbf{if} \text{sch1} = nil \textbf{then} (\text{return false})

\textbf{else} (\text{return sch1})

\textbf{fi}

\textbf{fi}

\textbf{SUM-WTS} \ <\text{class}\>\<\text{rel-op}\>\<\text{number}\>

This is the knapsack class of constraints. These constraints specify aggregate weight and plan object count requirements from some given class of plan objects discriminated on
5.3. CL Semantics

plan object attributes. They get tested and applied on a given state $S_i$. If the specified class is ALL then all the plan objects in $S_i.CS$ is considered. Tester function:

**test-wt-req**: $S \times CONST \rightarrow Cons-Sat$

Input: $S_i:S, C:CONST$

```
case C: equality
    if (sumwts (get-objects C.class (get-plan-objs $S_i.CS))) = C.number then
        (return true)
    elseif (sumwts (get-objects C.class (get-plan-objs $S_i.CS))) > C.number then
        (return false)
    else (return satisfiable)
    fi

case C: upper-limit
    if (sumwts (get-objects C.class (get-plan-objs $S_i.CS))) \leq C.number then
        (return true)
    else (return false)
    fi

case C: lower-limit
    if (sumwts (get-objects C.class (get-plan-objs $S_i.CS))) \geq C.number then
        (return true)
    else (return satisfiable)
    fi
```

Satisfier function:

**process-wt-req**: $S \times CONST \rightarrow S \cup \{fail\}$

Input: $S_i:S, C:CONST$, and a working set WS of plan objects.

```
case C: equality
    if (test-wt-req $S_i C) = true then
        (update-state $S_i C))
```
elseif (test-wt-req $S_i C) = false then
(return fail)

else (*$C$ is satisfiable.*)

\[
delta \leftarrow C\text{-number} - (\text{sumwts (get-objects } C\text{-class (get-plan-objs } S_i, CS)))
\]

obj-set \leftarrow (get-objects $C$\text{-class } WS) \setminus (get-plan-objs $S_i, CS$)

conseq-set \leftarrow (get-conseq-set $S_i, CS$)

(*Get the consequent set of all the plan objects in $CS$.*)

a-set \leftarrow (get-selection $\delta$ obj-set conseq-set (get-bias-seq))

(*Get an expansion for $C$. get-selection finds the promising set of plan objects to satisfy $C$ with respect to the consequent set of plan objects in existing $CS$ and a clustered knowledge structure derived from case data $LStruct$, get-bias-seq creates a sequence of plan objects as specified in user-given category preferences and is used in focusing selection of plan objects.*)

A: css-cand \leftarrow (local-select (dof (get-pi a-set)) $S_i, CS$)

(*css-cand is a set of conjunctive support structures which satisfies $C$ and form potential candidates.*)

new-sched \leftarrow (find-schedule css-cand $S_i$)

if new-sched \neq nil then
$S_{i+1}\cdot EX \leftarrow a$-set
(return $S_{i+1}$)

elseif new-sched = nil and more-expansions then
a-set \leftarrow (choose-next-expansion a-set $\delta$ obj-set conseq-set (get-bias-seq))
go to A
(*Nogoods database is also updated.*)

else (return fail)

fi

fi

case $C$: upper-limit

(*Using upper-limit constraints as generators.*)

if (test-wt-req $S_i C) = false then
(return fail)

else
\[ S \leftarrow C.\text{number} - (\text{sumwts (get-objects } C.\text{class (get-plan-objs } S_i.CS))) \]
\[ \text{obj-set } \leftarrow (\text{get-objects } C.\text{class } WS \setminus (\text{get-plan-objs } S_i.CS)) \]
\[ \text{conseq-set } \leftarrow (\text{get-conseq-set } S_i.CS) \]
\[ \text{a-set } \leftarrow (\text{get-selection } \delta \text{ obj-set conseq-set (get-bias-seq)}) \]

B: \ css-cand \leftarrow (\text{local-select (dof (get-pi a-set)) } S_i.CS) 
\[ \text{new-sched } \leftarrow (\text{find-schedule css-cand } S_i) \]
\[ \text{if } \text{new-sched } \neq \text{nil then} \]
\[ S_{i+1}.EX \leftarrow \text{a-set} \]
\[ (\text{return } S_{i+1}) \]
\[ \text{elseif}\ \text{new-sched } = \text{nil and more-expansions then} \]
\[ \text{a-set } \leftarrow (\text{choose-next-expansion a-set } \delta \text{ obj-set conseq-set (get-bias-seq)}) \]
\[ \text{go to B} \]
\[ \text{else (return fail)} \]

fi

\[ \text{case } C: \text{ lower-limit} \]
\[ \text{if} \ (\text{test-wt-req } S_i C) = \text{true then} \]
\[ (\text{return (update-state } S_i C)) \]
\[ \text{else } (*C \text{ is satisfiable.}*) \]
\[ \delta \leftarrow C.\text{number} - (\text{sumwts (get-objects } C.\text{class (get-plan-objs } S_i.CS))) \]
\[ \text{obj-set } \leftarrow (\text{get-objects } C.\text{class } WS \setminus (\text{get-plan-objs } S_i.CS)) \]
\[ \text{conseq-set } \leftarrow (\text{get-conseq-set } S_i.CS) \]
\[ \text{a-set } \leftarrow (\text{get-selection } \delta \text{ obj-set conseq-set (get-bias-seq)}) \]

C: \ css-cand \leftarrow (\text{local-select (dof (get-pi a-set)) } S_i.CS) 
\[ \text{new-sched } \leftarrow (\text{find-schedule css-cand } S_i) \]
\[ \text{if} \ \text{new-sched } \neq \text{nil then} \]
\[ S_{i+1}.EX \leftarrow \text{a-set} \]
\[ (\text{return } S_{i+1}) \]
\[ \text{elseif} \ \text{new-sched } = \text{nil and more-expansions then} \]
\[ \text{a-set } \leftarrow (\text{choose-next-expansion a-set } \delta \text{ obj-set conseq-set (get-bias-seq)}) \]
\[ \text{go to C} \]
\[ \text{else (return fail)} \]
This constraint also is a knapsack constraint. Here, number of plan objects are specified in the constraints rather than weights.

**Tester function:**

\[\text{test-count-objs}: S \times \text{CONST} \rightarrow \text{Cons-Sat}\]

**Input:** \(S, C\) as an instance of the above constraint \((C:\text{CONST})\)

\[\text{case } C: \text{ equality}\]

\[\begin{align*}
\text{ if } & (\text{count-objs (get-objects } C.\text{class (get-plan-objs } S_i.CS))) = C.\text{number} \text{ then} \\
& \text{ (return true)} \\
\text{ elseif } & (\text{count-objs (get-objects } C.\text{class (get-plan-objs } S_i.CS))) > C.\text{number} \text{ then} \\
& \text{ (return false)} \\
\text{ else } & \text{ (return satisfiable)} \\
\end{align*}\]

\[\text{fi}\]

\[\text{case } C: \text{ upper-limit}\]

\[\begin{align*}
\text{ if } & (\text{count-objs (get-objects } C.\text{class (get-plan-objs } S_i.CS))) \leq C.\text{number} \text{ then} \\
& \text{ (return true)} \\
\text{ else } & \text{ (return false)} \\
\end{align*}\]

\[\text{fi}\]

\[\text{case } C: \text{ lower-limit}\]

\[\begin{align*}
\text{ if } & (\text{count-objs (get-objects } C.\text{class (get-plan-objs } S_i.CS))) \geq C.\text{number} \text{ then} \\
& \text{ (return true)} \\
\text{ else } & \text{ (return satisfiable)} \\
\end{align*}\]

\[\text{fi}\]

**Satisfier function:**

\[\text{process-count-objs}: S \times \text{CONST} \rightarrow S \cup \{\text{fail}\}\]

**Input:** \(S, C:\text{CONST}\), and a working set \(WS\) of plan objects.

\[\text{case } C: \text{ equality}\]
if (test-count-objs $S_i C) = true then
(update-state $S_i C))

elseif (test-wt-req $S_i C) = false then
(return fail)

else (*$C$ is satisfiable.*)

$\delta \leftarrow C \cdot \text{number1} - \text{(count-objs (get-objects $C$.class (get-plan-objs $S_i.CS$)))}$

$\text{obj-set} \leftarrow \text{(get-objects $C$.class $WS$) \setminus (get-plan-objs $S_i.CS$)}$

$conseq-set \leftarrow \text{(get-conseq-set $S_i.CS$)}$

$a$-set $\leftarrow \text{(get-selection $\delta$ obj-set conseq-set (get-bias-seq))}$

D: $\text{css-cand} \leftarrow \text{(local-select (dof (get-pi a-set)) $S_i.CS$)}$

$\text{new-sched} \leftarrow \text{(find-schedule css-cand $S_i$)}$

if $\text{new-sched} \neq \text{nil}$ then
$S_{i+1}.EX \leftarrow a$-set
(return $S_{i+1}$)

elseif $\text{new-sched} = \text{nil}$ and more-expansions then
$a$-set $\leftarrow \text{(choose-next-expansion a-set $\delta$ obj-set conseq-set (get-bias-seq))}$

go to D

else (return fail)

fi

case $C$: upper-limit

if (test-count-objs $S_i C) = false then (return fail)

else

$\delta \leftarrow C \cdot \text{number1} - \text{(count-objs (get-objects $C$.class (get-plan-objs $S_i.CS$)))}$

$\text{obj-set} \leftarrow \text{(get-objects $C$.class $WS$) \setminus (get-plan-objs $S_i.CS$)}$

$conseq-set \leftarrow \text{(get-conseq-set $S_i.CS$)}$

$a$-set $\leftarrow \text{(get-selection $\delta$ obj-set conseq-set (get-bias-seq))}$

E: $\text{css-cand} \leftarrow \text{(local-select (dof (get-pi a-set)) $S_i.CS$)}$

$\text{new-sched} \leftarrow \text{(find-schedule css-cand $S_i$)}$

if $\text{new-sched} \neq \text{nil}$ then
$S_{i+1}.EX \leftarrow a$-set
(return $S_{i+1}$)
```
elseif new-sched = nil and more-expansions then
a-set ← (choose-next-expansion a-set δ obj-set consequ-set (get-bias-seq))
go to E
else (return fail)
fi

fi

case C: lower-limit

if (test-count-objs S_i C) = true then
(return (update-state S_i C))
else (*C is satisfiable.*)
δ ← C.number1 - (count-objs (get-objects C.class (get-plan-objs S_i.CS)))
obj-set ← (get-objects C.class WS) \ (get-plan-objs S_i.CS)
conseq-set ← (get-conseq-set S_i.CS)
a-set ← (get-selection δ obj-set consequ-set (get-bias-seq))
F: css-cand ← (local-select (dof (get-pi a-set)) S_i.CS)
new-sched ← (find-schedule css-cand S_i)
if new-sched ≠ nil then
S_{i+1}.EX ← a-set
(return S_{i+1})
elseif new-sched = nil and more-expansions then
a-set ← (choose-next-expansion a-set δ obj-set consequ-set (get-bias-seq))
go to E
else (return fail)
fi
fi

TAKE-OBJ-FROM <class> [deg-pref>
This inclusion constraint implicitly specifies inclusion of plan objects from a certain class. This class is discriminated using the pre-defined plan object attributes. We refer to this class of constraints (and their negations, see under exclusion constraints) as category-preference constraints.
```
The constraint specifies that in the selection of plan objects to satisfy the inclusion constraints, the objects that are to be chosen from the specified classes. \textit{deg-pref} gives the ordering on the selection; it is in order from most preferred to least preferred. The degree of preference parameter establishes a value ordering from possibilities prior to variable assignments in search. A bias sequence is created from \textit{WS} for a set of category-preference constraints. This is done the function \textit{get-bias-seq}. Plan objects get selected in order from this sequence. As each set gets exhausted, the following set in the sequence gets used.

For example, in degree planning, if \( C_1: \text{"take-obj-from computer-science most"}, \ C_2: \text{"take-obj-from mathematics most"} \) and \( C_3: \text{"take-obj-from geology"} \) are specified then a bias sequence of sets of units is created by \( \text{get-bias-seq} \ ((C_1 \ C_2) \ C_3) \). This returns the bias sequence: \(<\{\text{a random mixture of all computer-science and mathematics units}\}, \ \{\text{all geology units}\}>>\).

The category-constraints are also used in deriving matching plans from experiential knowledge. All those plans are selected which have the same ordering in weights from the plan object classes as specified in the category-constraints.

### 5.3.2.3 Load Constraints

The load constraints are \textit{upper limit} aggregate constraints which specify upper bounds on number of objects which can be placed in each temporal interval. They are local constraints and may also be used as inclusion constraints.

\textbf{SUM-WTS} \(<\text{temporal-interval}> \text{AT-MOST} <\text{number}>\)

\emph{Tester} function:

\textit{test-load-cons}: \( S \times \text{CONST} \rightarrow \text{Cons-Sat} \)

Input: \( S_i:S, \ C: \text{CONST} \)

\begin{verbatim}
    if (sumwts (get-objects C.temporal-interval (get-plan-objs S_i.CS))) \leq C.number
    then (return true)
    else (return false)
fi
\end{verbatim}

\emph{Satisfier} function for a load constraint \( C \) when it is used as a generator:

\textit{process-load-cons}: \( S \times \text{CONST} \rightarrow S \cup \{\text{fail}\} \)

Input: \( S_i:S, \ C: \text{CONST} \), and a working set \( WS \) of plan objects.
if (test-load-cons $S_i C$) = false then
  (return fail)
else
  (*Weight limit not reached and $C$ can be used as a generator.*)
  obj-set ← (get-objects $C$.temporal-interval $WS$) \ (get-objects $C$.temporal-interval (get-plan-objs $S_i.CS$))
  $\delta$ ← $C$.number - (sumwts (get-objects $C$.temporal-interval (get-plan-objs $S_i.CS$)))
  conseq-set ← (get-conseq-set $S_i.CS$)
  (*Get the consequent set of all the plan objects in $CS$.*)
  a-set ← (get-selection $\delta$ obj-set conseq-set (get-bias-seq))
  F: css-cand ← (local-select (dof (get-pi a-set)) $S_i.CS$)
  new-sched ← (find-schedule css-cand $S_i$)
  if new-sched ≠ nil then
    $S_{i+1}.EX$ ← a-set
    (return $S_{i+1}$)
  elseif new-sched = nil and more-expansions then
    a-set ← (choose-next-expansion a-set $\delta$ obj-set conseq-set (get-bias-seq))
    go to F
  else (return fail)
fi
fi

5.3.2.4 Exclusion Constraints

There are two kinds of exclusion constraints:

EXCLUDE <plan-objects>, and

NOT-TAKE-OBJ-FROM <class>

They specify certain objects or classes of objects which are not to be present in the plan being searched.

If all the exclusion constraints are known prior to the search, as is the case in Model-Search, then a working set $WS$ of Plan-Objs is made and the plan objects specified in the constraints are filtered from it (i.e., $WS \subseteq$ Plan-Objs). In IPLAN,
the exclusion constraints may get added during problem solving and they are reasoned with differently as we shall see in the next chapter.

Generally, the exclusion constraints are local constraints i.e., they get checked during the generation of each successor state in search.

*Tester* function:

\[
\text{test-exclusion-cons}: CSS \times CONST \rightarrow Cons-Sat
\]

Input: css:CSS, an exclusion constraint C:CONST, and a working set WS of plan objects.

\[
\begin{cases}
\text{if } (\text{excludep? } C) \text{ then } \\
\text{(*excludep? checks whether } C \text{ is an explicit exclusion constraint.*)} \\
\text{if } C.\text{plan-objects } \subseteq (\text{get-plan-objs css}) \text{ then } \\
\text{(return false)} \\
\text{else (return true)} \\
\text{fi} \\
\text{elseif } (\text{intersect (get-objects } C.\text{class } WS) (\text{get-plan-objs css})) \neq \emptyset \text{ then } \\
\text{(return false)} \\
\text{else (return true)} \\
\text{fi}
\end{cases}
\]

5.3.2.5 Temporal Constraints

Temporal constraints specify durations of the schedules and relative placements of plan objects into schedules. All the temporal constraints are processed in the scheduling process. The set of all temporal constraints form a subset of local constraints.

We present below the semantics of each category of the temporal constraints.

The following constraints are called the *plan-time-cons*:

\[
\begin{align*}
\text{PLAN-TIME } & \langle \text{number1}\rangle<\text{temporal-class}> \\
\text{PLAN-TIME-BETWEEN } & \langle \text{number1}\rangle<\text{temporal-class}> \text{ AND } \langle \text{number1}\rangle<\text{temporal-class}>
\end{align*}
\]

They specify time bounds on schedules.

The former constraint specifies the sequence of temporal intervals that is allowed for a schedule. For example, in degree planning, **PLAN-TIME 3 years** dictates that a degree schedule has to be searched which requires at most three years to complete.
Also, in degree planning, the condition may be specified in terms of number of sessions such as \texttt{PLAN-TIME 6 sessions}.

The latter constraint specifies the lower and upper time bounds on schedules. If a schedule conforming to the lower bound cannot be found then the lower bound is progressively relaxed and a schedule searched until a schedule is found or the upper bound is reached. This feature is currently missing in the CDSP problem solvers.

The constraints from \texttt{plan-time-cons} are used by \texttt{get-schedule} to create the schedule templates.

\textbf{TAKE-IN} \texttt{<plan-objects><temporal-interval>}

The above constraint is a \texttt{sched-pos-cons} constraint. It explicitly specifies intervals in schedules for placement of certain plan objects. It gets used both as a generator constraint and in the scheduling part of the problem solving.

In its use as a generator constraint, the presence of \texttt{C.plan-objects} in \texttt{Si} is tested by the \texttt{(test-obj-inclusion C)} function. If this fails then \texttt{(process-obj-inclusion C)} is invoked.

In scheduling, it is used by the \texttt{get-schedule} function. \texttt{get-schedule} takes a set of sets of well-formed temporal structures, \texttt{At}, and a set of temporal constraints, and finds a schedule if one exists. It creates a schedule template, uses any \texttt{sched-pos-cons} constraints in directing search for a combination of structures from each set in \texttt{At} such that all the temporal constraints are satisfied. The search space results from the availability constraints on plan objects.

\textbf{TAKE-CONC} \texttt{<plan-objects>}

This is a coincident constraint. It specifies that the set of plan objects given by the set \texttt{plan-objects} have to appear in the same temporal interval in the solution. It gets used both as a generator constraint and in the scheduling part of the problem solving.

In its use as a generator constraint, the presence of \texttt{C.plan-objects} in \texttt{Si} is tested by the \texttt{(test-obj-inclusion S; C)} function. If this fails then \texttt{(process-obj-inclusion S; C)} is invoked.

In scheduling, it is used by the \texttt{get-schedule} function. Prior to running the search for a consistent combination of the well-formed structures from \texttt{Ats}, all the sequences which do not have the objects given in \texttt{plan-objects} assigned to the same interval are filtered.

\textbf{TAKE-BEFORE} \texttt{<plan-object1><plan-object2>}

This is a precedent constraint. It specifies that for a schedule to satisfy this constraint it has to have the plan object given by \texttt{plan-object1} assigned in an interval succeeding
the interval to which the plan object given by \textit{plan-object}2 is assigned. The constraint gets used both as a generator constraint and in the scheduling part of the problem solving.

In its use as a generator constraint, the presence of the two plan objects given in the constraint in some given state \( S_i \) is tested by the (test-obj-inclusion \( S_i \ C \)) function. If this fails then (process-obj-inclusion \( S_i \ C \)) is invoked.

In scheduling, it is used by the \textit{get-schedule} function. Prior to running the search for a consistent combination of the well-formed structures from \( \text{Ats} \), all the sequences which have \textit{plan-object}1 assigned to the interval to which \textit{plan-object}2 is assigned are filtered.

### 5.3.2.6 Support Constraints

The support constraints are defined for plan objects. They are a mixture of inclusion constraints. Semantics of all the inclusion constructs in CL are given above.

### 5.3.2.7 Change Constraints

Once a solution is found, the user may request search for other solutions or he/she may request minimal changes to some existing solution at run-time. This is achieved through the issue of the following run-time constraints to the problem solver.

**FIND-ANOTHER-SOLN**

This constraint instructs the search engine to find another solution for an existing set of constraints. The existing solution is recorded and chronological backtracking invoked from the existing state in search for another solution. In the search for new solutions the already discovered solutions are avoided.

**SWAP<plan-object1><plan-object2>**

Given a \( S_i, \text{CS} \), \textit{plan-object}1 is removed from it and \textit{plan-object}2 is added with its supports. The search begins anew with EXCLUDE \textit{plan-object}1 and INCLUDE \textit{plan-object}2 added to the set of constraints.

**KEEP-ALL-BUT <plan-objects>**

This constraint specifies that of all the plan objects in \( S_i, \text{CS} \), filter those which are in \textit{plan-objects}. The resulting set of plan objects \( PO \) are treated as inclusion constraints: \( \forall \textit{po} \in PO \) INCLUDE \textit{po}, and the new set of constraints processed anew. This class of constraints allows the already developed solutions to be minimally perturbed in light of user directed “fine-tuning”.

CHANGE-CATEG-PREF

Any change in general category preferences is induced by this constraint. The system elicits the new preferences from the user and finds a plan anew choosing plan objects from the bias-sequence and consequent sets of the committed plan objects.

5.3.2.8 Conditional Constraints

The WHAT-IF constraints are the conditional constraints. In the Model-Search approach, they are treated as the normal constraints. In IPLAN they are used to answer conditional queries based on some problem solving state. They are used to explore potential paths to solutions from some point in the search space.
Chapter 6

Problem Solving Machinery: IPS

6.1 Introduction

In the previous chapter we introduced the problem solving language CL-IPS for the proposed CDSP model. The knowledge representation issues, and a problem input language with its semantics have been presented. This chapter presents the problem solving component IPS of CL-IPS.

IPS forms the heart of the CDSP problem solving model. It captures the characteristics of CDSP problems, including the desired features, and the problem structure of CDSP problems, as discussed in Chapter 3. In this chapter, a constraint satisfaction problem solving framework for planning as required by CDSP problems is developed in light of CDSP problem properties and the inadequacies of the existing CSP techniques (as discussed in Chapter 3). The CDSP problems are characterised and formulated as planning from "scratch" (i.e., Model-Search) problems. (The incremental planning formulation of CDSP problems, called IPLAN problems, is the subject of next chapter).

The current chapter is organised as follows. The common search approaches used in CSP problem solving of immediate relevance to CDSP problem solving are reviewed. CSP theory is extended to address the dynamic properties of CDSP problem solving. In particular, we address dynamic definition of constraint satisfaction parameters (viz., variables, variable associated domains and constraints) during search, selection (from a predefined partial ordering) of constraints and their effective use in focusing search, use of domain knowledge (such as heuristics drawn from experiential knowledge) in focusing search, and plan revisions through run-time constraints. The plan evaluation and plan verification problem solving components of CDSP problem solving are presented. Finally, the Model-Search formulation of CDSP problems is given. The Model-Search problem and its solution are characterised, and its solution procedure presented.
6.2 Search Approaches for Constraint Satisfaction

As we have seen in Chapters 2 and 3, a CSP = < V, D, C >, where V is a set of V_i variables, D is a set D_i of finite domains^1 associated with each variable V_i, and C is a set of constraints. A solution to a CSP is a set of assignments of the variables from V to values from D such that all the constraints in C are satisfied. In recent years, a variety of general approaches have been developed for finding assignment of values to a predefined set of variables such that all the constraints are satisfied. These are essentially search-based approaches. Some of these approaches have been reviewed in §2.4.2 of Chapter 2 (see page 18).

We now explain and illustrate the search approaches which form the basis for many problem solving models and which are of immediate relevance to the CDSP model. These approaches have been used in many planning and constraint satisfaction systems.

In the simple state space diagram of Figure 6.1, from state S_0, one might choose either S_1 or S_5. From S_1, one might choose either S_2 or S_4. There is only one choice from S_2. Hence, the respective choice points: from S_0 is \{S_1, S_5\}, from S_1 is \{S_2, S_4\}, and from S_2 is \{S_3\}. A search process chooses an alternative from each of the choice points until a solution (or all solutions) is found. A solution is a path through the state graph to a terminal state that satisfies the problem definition of a solution.

\begin{figure}[h]
\centering
\begin{tikzpicture}
\node (S0) at (0,0) {$S_0$};
\node (S1) at (-1,-1) {$S_1$};
\node (S5) at (1,-1) {$S_5$};
\node (S2) at (-1,-3) {$S_2$};
\node (S4) at (1,-3) {$S_4$};
\node (S3) at (0,-5) {$S_3$};
\draw (S0) -- (S1);
\draw (S0) -- (S5);
\draw (S1) -- (S2);
\draw (S1) -- (S4);
\end{tikzpicture}
\caption{A State-Based Search.}
\end{figure}

In a CSP, the set of choices of value assignments to variables at any point in problem solving constitutes a context (also called a problem state). Each new variable assignment defines a new context. For example, in Figure 6.2, variable V_1 is assigned the value a_1, then V_2 the value b_2, and to V_3 a value has not yet been assigned. The set

^1Note the domain for a variable in CSPs means a set of candidate values each of which can be assigned to the variable during search for constraint satisfaction.
6.2. Search Approaches for Constraint Satisfaction

\{V_1=a_1, V_2=b_2\} is a context. In the formulation of a CSP as a state-based search, in each state, a variable from a known set of variables is assigned to a value from a known set of possible values for the variable. For example, suppose there are three variables \{V_1, V_2, V_3\}. The domains for each of the variables are:

- \(V_1 \hookrightarrow \{a_1, a_2\}\)
- \(V_2 \hookrightarrow \{b_1, b_2, b_3\}\),
- \(V_3 \hookrightarrow \{c_1\}\), and
- \(V_4 \hookrightarrow \{d_1, d_2\}\). There will be twelve distinct assignments possible from these domains. In search, the state graph is traversed by assigning values from the choice points to the variables one at a time. From Figure 6.1, we could make the following correspondence.

- \(S_0\): no assignments
- \(S_1\): \(V_1 = a_1\)
- \(S_5\): \(V_1 = a_2\)
- \(S_2\): \(V_1 = a_1 \land V_2 = b_1\)
- \(S_4\): \(V_1 = a_1 \land V_2 = b_2\)
- \(S_3\): \(V_1 = a_1 \land V_2 = b_1 \land V_3 = c_1\).

The context in \(S_3\) is \(\{V_1 = a_1, V_2 = b_1, V_3 = c_1\}\).

In a CSP, the assignments proceed as above and the contexts are checked against the constraints. If a constraint is violated by a context, the context becomes inconsistent and a backtracking search strategy is employed to systematically explore other options until a solution (i.e., a consistent context with all the variables assigned) is found or all the options are exhausted.

6.2.1 Standard Backtracking

Standard backtracking [72] (also known as just backtracking) is an important technique for solving CSPs. It is a search control strategy which combines depth-first
search with chronological backtracking and is endemic to PROLOG. It is chronological in that from the inconsistent context the control is transferred back to the previous (consistent) context and the most recent assignment is removed. Search re-starts with another assignment from this point. This way, any subgraphs leading from the inconsistent context gets pruned. This behavior results from the rule that if a context $CN_i$ is inconsistent, then neither is any other context $CN_{i+1}$ resulting from an extended sequence of choices. Thus, for such a $CN_i$, by induction, no extended context $CN_n$ with all the variables assigned will be consistent. Thus, backtracking is a good search strategy for CSPs because search need not continue to a terminal node to know whether a path does not lead to a solution.

For example, in the above assignments, let us suppose that the context in $S_2$ is inconsistent. The search is backtracked to $S_1$, and $V_2$ is assigned the value $b_2$. The resulting state is $S_4$. Such an assignment switch is called a context switch.

Although standard backtracking does improve on simple generate-and-test by pruning significant portions of search space, it still requires exponential time in the number of variables in the worst case [95]. Therefore standard backtracking as a control mechanism is not adequate for many real-world problems including the CDSP type problems.

Below, we give a standard backtracking algorithm from [121]. We later adapt this algorithm to make it "informed" for its use in the CDSP model.

**Standard Backtracking Algorithm**

1. Put the start node $S$ on OPEN. (*OPEN is a stack*.)
2. If OPEN is empty, exit with failure, otherwise continue.
3. Examine the topmost node from OPEN and call it $n$.
4. If all the successors of $n$ have been generated, remove $n$ from OPEN and go to step 2.
5. Generate a successor-the $(n + 1)$th node. Call this $n'$. Put $n'$ on top of OPEN and provide a pointer back to $n$.
6. Mark $n$ to indicate the path ($n$, $n'$) as traversed.
7. If the path ($S$, $n'$) results in the solution, exit with the it; otherwise continue.
8. If $n'$ is a dead end, remove $n'$ from OPEN.
9. Go to step 2.
Standard backtracking is complete (i.e., it finds a solution if there exists one) but has several drawbacks. In addition to the inherent pathological behavior (see Chapter 2, §2.4.2), the algorithm does not record any reasons for why or how choices were made [43]. Hence, it has no access to such information if backtracking becomes necessary\(^2\).

Several preprocessing techniques (such as variable ordering\(^3\), value ordering, forward checking, constraint recording and backjumping [38]) have been developed to make backtracking informed and efficient. These techniques are appropriately used later to make the above backtracking algorithm “informed” for use in the proposed CDSP model.

### 6.2.2 Dependency-Directed Backtracking

A more efficient way to recover from inconsistency is to focus only on the choices that matter i.e., the culprits which cause the inconsistencies (or the assignments on which the untenability ultimately depends). It is much more efficient to avoid undoing state searches unnecessarily. These features are realised in a progenitor search method of chronological backtracking called dependency-directed backtracking [153] (DDB). DDB is a kind of selective backtracking approach in which search is backtracked to the most recent choice that contributes to an inconsistency, the culprit choice is undone and an alternative choice is made. The search resumes from this point.

To illustrate the motivation for DDB, assume the above variable and the associated domain definitions. In addition, assume that none of the values in the domain for \(V_3\) are compatible with the values \(b_1\) and \(b_2\) of \(V_2\). Also, assume that all the values in the domain of \(V_4\) conflict with the choice of \(V_1 = a_1\). In the standard backtracking, the search would proceed as follows.

\(^2\)Kumar [91] reviews in detail the problems in backtracking.

\(^3\)A better ordering on variables for the above example, based on cardinality of the variable associated domains, is \(<V_3, V_4, V_1, V_2>\).
The resulting context after Step 7 is \( \{V_1 = a_1, V_2 = b_3, V_3 = c_1, V_4 = d_2\} \). This is inconsistent and backtracking would proceed to undo the \( V_1 = a_1 \) assignment and re-start the search with \( V_1 = a_2 \). This will re-discover the other incompatibilities of \( c_1 \) with \( b_1 \) and \( b_2 \).

In contrast, DDB preserves the current assignments and caches the known culprits as nogoods. This allows futile traversal of search space and thrashing to be avoided, and permits search to be complete. The nogoods is a set of smallest sets of assignments of variables each of which is known from experience to lead to an inconsistency. Suppose that \( V_1 = a_1 \) is replaced by \( V_1 = a_2 \). Using DDB the resulting context becomes \( \{V_1 = a_2, V_2 = b_3, V_3 = c_1, V_4 = d_2\} \). The assignments of \( V_2, V_3 \) and \( V_4 \) remain unchanged. This way, DDB jumps to a new problem solving state which has not been visited before. The resulting nogoods set from the above sequence of variable assignments is \{\( \{V_2 = b_1, V_3 = c_1\}, \{V_2 = b_2, V_3 = c_1\}, \{V_4 = d_1, V_1 = a_1\}, \{V_4 = d_2, V_1 = a_1\}\}\).

Formalisms for dependency-directed backtracking (such as Doyle’s TMS) use elaborate data structures to succinctly maintain the justification (or dependency) structures for choices explicitly. Information captured in these structures serve to facilitate backtracking.

A Dependency-Directed Backtracking Algorithm

Implementations of DDB in different formalisms differ in the use of data structures and logics. Below, we present a general DDB algorithm.

Whenever an inconsistent context \( CN \) arises in some state \( S_i \),

1. Add \( CN \) to \( NOGOODS \).

2. Trace back through the justifications of the assignments. Find and order the culprit assignments from most recent to least recent. Store it as the sequence of culprits \( CUL \).
3. Get the first culprit $cul$ from $CUL$ and remove $cul$ from $CUL$.

4. Undo the assignment in the $cul$ and assign an alternative value to the variable such that the resulting context does not contain an element from $NOGOODS$. If this is not possible then go to step 3. Otherwise, the updated state $S_{i+1}$ is assigned the context $CN_{i+1}$, continue.

5. Move forward again, making a choice for a new (i.e., unassigned) variable such that the context in the next state does not contain any nogoods.

DDB has been closely related to truth maintenance systems (TMSs) and is a central mechanism of Doyle’s [49] and McDermott’s [102] TMSs. It is suited to the CDSP type of problem solving. The nogoods feature is used in both Model-Search and IPLAN. IPLAN uses DDB in conjunction with an assumption-based truth maintenance system.

6.3 CDSP Problems and Constraint Satisfaction

As we have seen in Chapter 3, CDSP problems can be formulated as constraint satisfaction problems albeit inadequately. Recalling briefly, CSPs assume fixed variables, domains and constraints [106] whereas in CDSPs, all the variables and their associated domains, and the constraints are usually not known at the start of problem solving. Also, in CDSPs, the number of objects to be included in a plan is not usually known in advance. Hence, CDSP problems are of a dynamic nature where any of these parameters of the constraint problem (i.e., variables, domains, constraints) might change as the search progresses. We have also seen in Chapter 3 why the existing attempts at modeling dynamic constraint satisfaction problems cannot be used for solving CDSP problems.

The other drawbacks are that the CSP paradigm does not exploit distinctions between kinds of constraints in solving CSPs [124]. Also, a CSP formulation usually prescribes that all the constraints be extensional i.e., discrete, finite sets of values be defined for each variable whereas CDSP problems may have both extensional and intensional constraints.

Below, we discuss the constraint satisfaction properties of CDSP problem solving. These properties are later exploited in the computational schemes for the CDSP problem formulations.
6.3.1 Constraint Distinctions

Constraints are usually viewed as Boolean expressions (i.e., either an object satisfies a constraint or not). In CSP problem solving, constraints do not get differentiated and are used as Boolean expressions on objects [124]. The various kinds of constraints in CDSPs and the various constraint-related CDSP properties permit distinctions between constraints and their effective use in search. In addition to the Boolean semantics of the constraint expressions, the information contained in constraint expressions in the various classes can be used in guiding search in a similar tradition to the use of heuristics in guiding search. The functional classes of constraints have been introduced earlier in the thesis. Here, we summarise them. Later, in the algorithms for Model-Search and IPLAN, it will be shown how these different classes of constraints are effectively used in the search process.

![Figure 6.3: Summary of Constraint Categories.](image)

The Venn Diagram of Figure 6.3 summarises the major categories of constraints. (Note, in the figure, the sizes of the circles are arbitrarily chosen. They do not represent actual sizes of the sets.) In addition, Sched-Cons is a set of load, plan-time, precedent, coincident, sched-pos-cons and availability constraints; and the upper-limit, exclusion and incompatibility constraints make up the local selection constraints (as discussed in §5.3.2).
6.3. CDSP Problems and Constraint Satisfaction

In search processes of Model-Search and IPLAN, the exclusion constraints are used to filter the plan objects from the set of all plan objects in the domain. These plan objects may be explicitly or implicitly specified in the exclusion constraints. The inclusion constraints are used as the generator constraints. These constraints are systematically used to elucidate plans by selecting minimal sets of plan objects which satisfy them and including these plan objects with their supports (as conjunctive support structures) in the plans. Those support structures are chosen which are consistent with some existing LSC and which are consistent with Sched-Cons (i.e., schedulable).

6.3.2 Constraint Variables and Domains

Unlike CSPs, in CDSP problems, we do not know all the constraints, variables and values that each variable may assume at the start of problem solving. Nor do we know how many plan objects will constitute a plan. Although most of the formal constraints are usually known prior to problem solving, the support constraints for plan objects emerge during problem solving. Also, preferential constraints may be added and removed during problem solving.

To address the above drawbacks in CSP formulation of CDSP problems, we extend the CSP framework by accommodating dynamic creation of a variable and its associated domain for each constraint as it gets used in search. Unlike CSP, not all these variables have to be assigned a value to solve a problem. Satisfaction of some constraints may result in satisfaction of others or it may take certain other constraints closer to satisfaction.

In search, a finite number of states is systematically explored until a sequence \(<S_0, ..., S_i, ..., S_n>\) is found in which all the constraints in \(S_n.AC \cup \Delta\) are satisfied. Each state \(S_{i+1}\) is generated from \(S_i\) by selecting an inclusion constraint \(C\) and applying its satisfier operator \(\gamma_C\) to \(S_i\). \(\gamma_C\) is a macro-operator combining operators for making value choices and for testing local constraints for each choice made. Its application to \(S_i\) finds, if possible, a \(S_{i+1}\) in which there is a schedule which is consistent with all the constraints in \(S_i.AC \cup \Delta\) (including the processed constraints kept as \(S_{i+1}.PC\)).

In the formulation of CDSP problems as CSPs, a variable \(X\) is dynamically defined for each inclusion constraint \(C\). A variable has associated with it a domain \(D\) which is also dynamically created during search. \(D\) is a set of conjunctive support structures. Application of an inclusion constraint \(C\) (which is not already satisfied in \(S_i\)) to a state

\[\text{Note a schedule is consistent iff it is satisfiable i.e., if an interpretation of a set of constraints can be found which satisfies each of the constraints in the set } S_i.AC \cup \Delta.\]
$S_i$ would result in computation of the domain for $C$ with respect to $S_i$ and search for a conjunctive support structure $CSS_x$ from $D$ which, when added to $S_i.CS$ would result in a $CS$ that is consistent with $S_i.AC \cup \Delta$. If such a $CS$ can be found then $S_{i+1}$ is created with $C$ and $CSS_x$ assigned to the variables $S_i.C$ and $S_{i+1}.X$ respectively.

In search, $D$ is dynamically created based on the combination of previous variable assignments—$S_1.x \oplus \ldots \oplus S_i.x$, where $x$ denotes a $CSS$ value assigned to a variable $X$—as recorded in $S_i.CS$. It is derived by the constraint satisfier function for $C$ and forms a choice-point in search. If, however, a $D$ is empty then $C$ is not satisfiable by any extension to $S_i.CS$ and dead-end is reached in search.

In summary, dynamic creation of domains and variables for each constraint

- addresses the unknown constraints, variables and domains problems in CDSPs,
- makes the number of plan objects to be included in a plan an unimportant concern in search,
- allows an informed creation of domains for constraints based on knowledge in previous states in search,
- has the value propagation feature in variable assignments which allows values previously assigned to variables to be available for satisfaction of new constraints.

The last feature addresses the property of CDSP problems in which satisfaction of some constraints may satisfy or carry certain other constraints closer to satisfaction (i.e., constraints may be coupled), variables may share values, and it is imperative that all objects in a plan are unique. In CDSP search, previous variable assignments influence any future variable assignments.

### 6.4 Constraint Relationships

In addition to extending the CSP theory in the above ways and the use of information contained in constraints in search reduction, the CDSP problem solving framework exploits constraint relationships to effectively organise and consequently, further reduce search. There are two kinds of useful orderings typically found in CDSP problems: partial ordering on functional categories of constraints as discriminated above, and partial ordering on constraints based on specificity in the inclusion category. Also, some constraints may couple some other constraints. Selection and expedient use of constraints in search exploiting these relationships are important problem solving features intrinsic to both the Model-Search and IPLAN approaches. Below, we present
these relationships and explain how they are used in search. These relationships are abstracted from and discussed in terms of the constraint constructs in CL.

6.4.1 Ordering in Constraint Categories

CDSP problem solving requires dynamic constraint satisfaction of new constraints which emerge as search progresses. In the CDSP model, we select and apply constraints to points in search space systematically such that search is effectively and optimally reduced. The selection of constraints for application is tied to an ordering on constraint categories which we have identified in §6.3.1.

Given a set of all possible plan objects, Plan-Objs, first a working copy WS is made. Then, the exclusion constraints are used to filter all those plan objects from WS which cannot be included in solutions. The category-preference constraints (for example, “take-obj-from computer-science most” meaning it is most preferred to select units from computer-science) are then used to order plan objects for selection based on degrees of preferences. The inclusion constraints are then applied in an order heuristically determined in terms of the constraint constructs in CL (see §6.4.2 below). Application of each inclusion constraint involves a selection of a conjunctive support structure for a set of plan objects which satisfies the inclusion constraint. The chosen conjunctive support structure has to be consistent with exclusion constraints. It also has to result in a CS (by its addition to an existing CS) which is consistent with the local constraints: the incompatibility, upper-limit, equality and temporal constraints.

Hence, in the pre-processing step, the exclusion constraints are used for domain filtering and the category-preference constraints are used for value ordering prior to selecting plan objects for satisfying the inclusion constraints. In search, the inclusion constraints get applied through their tester and satisfier functions and the incompatibility, upper-limit and temporal constraints are used in local search. The support constraints are satisfied in the conjunctive support structures per se.

The use of the different categories of constraints in the above manner is characteristic of CDSP problem solving. The category ordering is predefined in terms of the constructs in CL.

6.4.2 Ordering in Inclusion Constraints

During search for CDSP problem solving, preferential constraints get added or retracted, and support constraints also get added through constraint propagation. To make search effective and efficient, we need to opportunistically and dynamically choose
Chapter 6. Problem Solving Machinery: IPS

inclusion constraints for application in search. This is possible through metrics on constraint specificities or strengths. (By an *opportunistic* choice we mean the choice of the most constraining or "strongest" constraint with respect to some state. Selections of weaker constraints are deferred until they become strong with respect to current point in search.) Automatic ordering and selection of inclusion constraints for their effective use in CDSP problem solving is a difficult problem. It is difficult because the constraints can be quite complex and their relative specificities may not be easily calculable. Hence, heuristic-based ordering and selection have to be used.

The specificity ordering can be defined in terms of the syntactic constructs for constraints in CL and also, in terms of the semantics of constraints, such as cardinalities of domains for constraints. A characteristic feature of CDSP problems that is useful in selection of constraints is that, almost all constraints specify temporal restrictions either explicitly or implicitly and constraints on latter temporal intervals are usually more constraining than constraints on earlier intervals. As we have seen in Chapter 3, the support constraints demand existence in a plan, of plan objects which satisfy them. The support constraints also implicitly require that certain temporal orderings between plan objects (as captured by conjunctive support structures) be met. As a result of this property, usually, plan objects which are available in latter temporal intervals have more constraining support constraints i.e., depths of their conjunctive support structures are higher than those of conjunctive structures for plan objects which are available in earlier temporal intervals. We call this the *constraining property of supports*. This property is fundamental to the constraint ordering and selection process and hence, an algorithm to choose a constraint has to realise the importance of this property. Below, we identify the syntactic and semantic classes of inclusion constraints in CDSP problems and demonstrate how their intrinsic ordering and other heuristics can be used to select best possible constraints during search. Later, in §6.5, we present an algorithm which takes a set of constraints and some problem solving state $S_i$, and finds an optimal constraint for application based on the state and the constraint ordering.

1. Type-A: Non-Knapsack Constraints

The non-knapsack inclusion constraints (which are also called non-limit constraints) have the syntax: INCLUDE <plan-object>. This construct expresses the most specific class of constraints. The expansion-sets for constraints in this class contain just one element, the plan object given in *plan-object*. From a set of these constraints, the constraint which specifies the plan object which is available in the earliest temporal interval is chosen. If there is more than one such con-
6.4. Constraint Relationships

Constraint then the choice is made arbitrarily. The reason for choosing constraints on earlier intervals first is that plan objects from earlier intervals often form pre-requisites to plan objects from latter intervals and the addition of plan objects and their supports from the earlier intervals may restrict choice of supports for plan objects from the latter intervals. This restriction is caused if the chosen plan objects from earlier intervals form supports for objects from latter intervals.

2. Type-B: Lower-Limit and Equality Constraints

The set of lower-limit and equality constraints which are given in terms of temporal intervals are identified and ordered with respect to their specificities as follows. All those constraints which specify the latest temporal interval are first found. A constraint which is most constraining is then found from these constraints. This is determined with respect to the cardinalities of the respective constraint domains. (We present below, in §6.5.1, rules for determining the most constraining constraints.) In the cardinality-based selection, the constraint with lowest domain cardinality is most constraining (or strongest) and gets selected.

3. Type-C: Non-Temporal Interval Constraints

The constraints which are not specified on temporal intervals but impose plan object category restrictions (for example, “Sum-Wts science at-least 12”—the category restriction is that the units chosen to satisfy this constraint have to be science units) restrictions are then ordered from most to least constraining in terms of their domain cardinalities—as in the case of Type-B constraints above.

4. Type-D: Load Constraints

The load constraints can be used as generators in search if application of the constraints in the above groups do not result in a consistent plan. In the constraint selection process, the load constraint on the latest interval is chosen first. This is determined by checking the load constraints against a pre-defined load constraint sequence. This sequence incorporates any composite intervals in positions immediately after their component intervals in the sequence. For example, in degree planning, we have load constraints given for each temporal interval and ordered from latest to earliest intervals as <year3session2, year3session1, year3, year2session2, year2session1, year2, year1session2, year1session1, year1>. The upper load limits defined for each of the intervals at the ANU for BSc degrees are

\begin{align*}
\text{year3session2: } & 3, \text{ year3session1: } 3, \text{ year3: } 6; \\
\text{year2session2: } & 3, \text{ year2session1: } 3, \text{ year2: } 6; \text{ and}
\end{align*}
5. Type-E: Constraints on All Objects
Finally, the constraints which specify restrictions on all plan objects are used as generator constraints. These constraints are not interval-based nor have category restrictions. An example from the degree planning domain is “Sum-Wts all at-least 20”.

6.4.3 Constraint Coupling
Constraints in CDSP problems may couple (or overlap). That is, satisfaction of some constraints may satisfy some other constraints or it may take some other constraints closer to satisfaction. The domains of coupled constraints overlap. If (expand $C_1$) $\subseteq$ (expand $C_2$) then $C_1$ subsumes $C_2$. For example, in degree planning, given the constraints $C_1$: “include at-least 4 points of science”, $C_2$: “include at-least 2 points of mathematics”, and that mathematics is a science subject then satisfaction of $C_2$ will move $C_1$ closer to satisfaction. $C_2$ subsumes $C_1$.

In general, if a constraint $C_x$ subsumes another constraint $C_y$ then the search process is more efficient if $C_x$ is applied prior to $C_y$. Application of $C_x$ prior to $C_y$ will also satisfy $C_y$. Application of $C_y$ prior to $C_x$ may result in selection of values which may not satisfy $C_x$ and hence is a source of inefficiency in search. The subsumption criterion for finding and selecting coupling constraints is captured in the proposed general constraint selection algorithm presented in the next section.

6.5 Constraint Selection
Before we present a constraint selection algorithm, we explain how domains and their cardinalities are computed for inclusion constraints with respect to a state in problem solving, and how the cardinalities are used in determining the most constraining constraints with respect to the state.

6.5.1 Search for Most-Constraining Constraints
Given a state $S_i$ and a working set $WS$ of plan objects, the estimates on cardinalities are obtained by finding the set of all plan objects derived from $WS$ and not already contained in $S_i, CS$, such that any attribute restrictions specified in the constraints are

\(^5\)Note (expand $C$) finds a set of sets of plan objects each of which satisfies the constraint $C$, see page 92, Chapter 4.
6.5. Constraint Selection

met. We call this set of plan objects for each constraint as a category-domain \( CD \) for \( C \). For example, in degree planning, the constraint “sum-wts science at-least 12” has a \( CD \) which is a set of all the science units from \( WS \) which are not in \( S_i.CS \). In general, \( CD \) for a knapsack constraint “Sum-Wts \{attribute restrictions\} <rel-op> <number>” is a set of all those plan objects which meet the attribute restrictions specified in the constraint.

The cardinality of the domain for each constraint \( C \) is found by

\[
\text{choice}(a; b) = \frac{a!}{(a - b)! b!}
\]

where \( ! \) denotes factorial of a positive integer, \( a \) and \( b \in \mathbb{N} \), \( a \) is the cardinality of the category-domain, \( CD \), for \( C \), and \( b \) is the number of plan objects that are to be selected in each combination to satisfy \( C \). For the constraints which specify restrictions on counts of plan objects, calculations of cardinalities of constraint domains are straightforwardly done using the above formula with \( b \) assigned \( C\cdot\text{number} \). This is because the counts are uniform. In the cases where plan objects have variable weights, and constraints specify restrictions on aggregate weights from different categories of plan objects, however, the choice formula has to be adapted in the following manner. For some constraint \( C \), given a set of possible weights \( W_t \) of plan objects in \( CD \) of \( C \), we find a weighted average \( Av \) of the weights of plan objects in a \( CD \), then take the weight \( w \) from \( W_t \) which is closest to \( Av \), and use it in the cardinality calculation in the following way:

\[
\text{choice}(a; b) = \frac{a!}{(a - b)! b!}
\]

where \( a \) is as given above and \( b' = \frac{C\cdot\text{number}}{w} \). For \( C \) to be satisfiable, \( b' \leq a \). This specificity-based selection of inclusion constraints allows selection of most constraining constraints at various points in search. Selection of weaker constraints are deferred until they become most constraining during search.

The following algorithm for the function \( \text{get-strong-cons} \) finds the most constraining constraint from a given set of constraints and a cumulative structure.

\( \text{get-strong-cons}: P \ \text{CONST} \times \text{CSS} \rightarrow \text{CONST} \)

Input: \( \text{cnsts} \subset \text{CONST}, CS:\text{CSS} \)

\[
\text{cons-card} \leftarrow 0
\]

\[
\text{for each } cns \in \text{cnsts}
\]

\[
\text{card} \leftarrow (\text{count-plan-objs} (\text{get-categ-domain} cns (\text{get-plan-objs} CS)))
\]
Chapter 6. Problem Solving Machinery: IPS

6.5.2 Constraint-Select Algorithm

The following constraint selection algorithm (called constraint-select) takes a set of inclusion constraints expressed in CL and a cumulative structure cs from an existing state $S_i$ in search, and finds, using heuristics, the best possible constraint suitable for application in search based on the orderings given above. The selection is based on information contained in the existing state and hence, at different points in search, constraint orderings may be different for the same set of constraints. Therefore, constraint-select has to be invoked at each new state in search.

constraint-select: $P\ CONST \times CSS \rightarrow CONST \cup \{\text{nil}\}$

Input: $Inc-Conss \subset CONST, cs:CSS$

\begin{verbatim}
if Inc-Conss = {} then (return nil) fi

for each $c \in Inc-Conss$

if (satisfied? $c\ cs$) then (return $c$) fi

endfor

(*satisfied? checks whether a cumulative structure satisfies a constraint.*)

if $\#Inc-Conss = 1$ then (return (choose Inc-Conss)) fi

(*If there is only one constraint to choose from Inc-Conss then return it.*)

 Initialise Type-A-Set, Type-B-Set, Type-C-Set, Type-D-Set, Type-E-Set, Unknown-Set to {}.

(*The inclusion constraints are categorised into different types, see §6.4.2.*)

for each $c \in Inc-Conss$

if $c$:Type-A then Type-A-Set $\leftarrow$ Type-A-Set $\cup \{c\}$
\end{verbatim}
else if c: Type-B then Type-B-Set ← Type-B-Set U \{c\}
else if c: Type-C then Type-C-Set ← Type-C-Set U \{c\}
else if c: Type-D then Type-D-Set ← Type-D-Set U \{c\}
else if c: Type-E then Type-E-Set ← Type-E-Set U \{c\}
else Unknown-Set ← Unknown-Set U \{c\}
fi

endfor

if Type-A-Set ≠ \{\} then (return (semantic-sort Type-A Type-A-Set cs))
else if Type-B-Set ≠ \{\} then (return (semantic-sort Type-B Type-B-Set cs))
else if Type-C-Set ≠ \{\} then (return (semantic-sort Type-C Type-C-Set cs))
else if Type-D-Set ≠ \{\} then (return (semantic-sort Type-D Type-D-Set cs))
else if Type-E-Set ≠ \{\} then (return (semantic-sort Type-E Type-E-Set cs))
else (return (semantic-sort unknown Unknown-Set cs))
fi

The semantic-sort function takes one of the above inclusion constraint types Cons-Type, a corresponding set of constraints for each type and a cumulative structure from existing state, and finds the optimal constraint for application with respect to the existing state.

**semantic-sort:** Cons-Type \times P \text{CONST} \times \text{CSS} \rightarrow \text{CONST}
**Input:** cons-type: Cons-Type, cnsts \subset \text{CONST} and cs: CSS

**case** cons-type: Type-A (return (get-lowest-interval-constraints cnsts))
(*Find the constraint from cnsts which specifies plan objects for inclusion from the lowest possible temporal interval. This permits inclusion of supports for higher level plan objects specified in other inclusion constraints to be informed.*)

**case** cons-type: Type-B

cons-set ← (find-highest-int-cons cnsts)
(*Find the highest interval specified in the constraints in cnsts.*)
(return (get-strong-cons cons-set cs))

(*Find and return the most constraining constraint from the constraint set on the highest interval using the domain cardinality ordering.*)

case cons-type: Type-C
(return (get-strong-cons cnsts cs))

(*Find, using the cardinality ordering heuristic, the most constraining constraint from cnsts with respect to cs.*)

case cons-type: Type-D
(return (get-highest-interval-load-cons cnsts))

(*Find the load constraint on the highest interval of all the load constraints given in cnsts.*)

case cons-type: Type-E
(return (get-highest-interval-load-cons cnsts))

(*Find the load constraint on the highest interval of all the load constraints given in cnsts.*)

otherwise (return (ask-user-for-selection cnsts cs))

(*If there are constraints which do not fit in the above categories, display cs and the constraints, and ask the user to choose one.*)

6.6 Essential Components in Problem Solving

Prior to giving the formulations of CDSP problems and their solution procedures, we give below the problem solving components that are commonly required by the solution procedures for each of the formulations.

6.6.1 Problem Specification and Input

In the CDSP model, all the background knowledge are kept in a data file and managed by the KMS component. The COBWEB clusters get updated against current formal constraints regularly by the CDSP-COBWEB component. There are two other databases: one for plans and the other for executed plans. This knowledge base is made available to the CDSP problem solver. As for the dynamic knowledge, CL allows the communication of preferential constraints to the problem solver. Also, the various different components of the system are interactively invoked by the user through CL. At the end of a planning session, the system saves an acceptable schedule, its associated
set of preferential constraints and cumulative structure into a plan database which is saved as a file—the reason for retaining the cumulative structure is that because they capture the support knowledge, they facilitate revision in the event that the schedule is found to be inconsistent with formal constraints in the future. This information is uniquely identified by a plan id. It gets retrieved for later use. Regularly, the schedule is verified against formal constraints, and updated in light of any inconsistencies arising with changing formal constraints. Also, the schedule is revised in light of changing preferential constraints. Any arising inconsistencies with constraints are resolved by re-planning from the existing state as we shall see later in the solution procedures.

As plans are executed, the results are recorded in a database of XPLANS which are identified with the corresponding plan ids. An xplan captures, among other attributes, the plan execution start time, and a set of tuples each containing the plan object executed, the result of its execution and the interval in which it was executed (see Chapter 4, page 85 for the complete definition).

### 6.6.2 Plan Evaluation

To check whether an executed plan has satisfied all the constraints to qualify as a successful plan (i.e., whether the plan is valid with respect to an existing set of formal constraints), it has to be evaluated against an existing set of formal constraints. The evaluation requires checking satisfaction of each of these constraints and any constraints on the results of execution of the plan, called result-cons. (The minimum performance constraints in the degree planning domain are examples of result-cons constraints.) The checking is performed by the appropriate tester functions of the constraint objects. If the plan is found not to be consistent with the constraints then the violated constraints are returned. Otherwise, pass is returned denoting the executed plan is valid. Plan evaluation is performed by the following function:

\[
\text{evaluate-plan: XPLANS} \rightarrow \text{P CONST} \cup \{\text{pass}\}
\]

Note, in degree planning, the formal constraints change often. The changes are usually enforced on new plans. The set of constraints in force at the time of a degree enrolment is taken to be the set of constraints active on the degree plan until its total execution. If, however, there are any changes in the formal constraints during the course of a plan execution, it is ensured that the changes do not lead to inconsistency with the existing set of formal constraints. In the evaluation of degrees, therefore, the degrees are checked against the formal constraints active at the time of evaluation.
6.6.3 Plan Verification and Repair

It is important to regularly verify and, if required, revise a previously accepted plan. Verification is necessary because of the dynamic nature of formal constraints and other planning knowledge, and any contingencies. For example, in the degree planning domain, because degree requirements change often, it is imperative to ensure that the plan retained is consistent with new requirements. Also, the student might fail units prompting re-planning. Typical changes in CDSP knowledge include:

- extension to the set of plan objects by the addition of new plan objects,
- deletion of plan objects from plan object sets,
- changes in support requirements,
- changes in load restrictions on temporal intervals,
- plan time changes, and
- changes in other formal constraints.

The verification of a plan requires checking whether it satisfies an existing set of formal constraints and whether a plan execution is “going according to plan”. Violation of at least one constraint renders a plan inconsistent. The following function takes a schedule and the corresponding executed plan, performs the verification on it, and returns true if the schedule is consistent, otherwise it returns false. Note, the formal constraints are globally defined as Form-Cons.

verify-plan: Schedules × XPLANS → Boolean
Input: sch:Schedules

if (not (satisfies-preqs? sch)) then (return false) fi
(*All the objects in the schedule must have prerequisites in the plan, and they must appear temporally preceding the respective objects in the schedule.*)

if (not (satisfies-coreqs? sch)) then (return false) fi
(*If any object in the schedule requires a co-requisite then the co-requisite must also be in the schedule, and it has to appear in the same interval as that of the object or in an interval preceding the interval of the object.*)

if (not (incomp-inconsistent? (get-plan-objs sch) (get-incomp-cons Form-Cons))) then (return false) fi
6.6. Essential Components in Problem Solving

(*If there are incompatible objects in the schedule then the schedule is inconsistent.*)

\[
\text{if (not (satisfies-knapsack? (get-plan-objs sch) (get-knapsack-cons \textit{Form-Cons})) then (return false) fi)
}\]

(*If any one of the knapsack constraints are violated then the schedule is inconsistent.*)

\[
\text{if (not (satisfies-obj-avail? sch)) then (return false) fi}
\]

(*If there is an object appearing in an interval different to the intervals it is available in then the schedule is inconsistent.*)

\[
\text{if (not (satisfies-temp-cons? sch (get-temp-cons \textit{Form-Cons}))) then (return false) fi}
\]

(*If at least one of the temporal constraints is not satisfied then the schedule is inconsistent.*)

\[
\text{if (incompat-plan-xplan? sch xplan) then (return false) fi}
\]

(*If some objects from xplan are not countable and they also belong to the schedule, then the plan and the executed plan are not compatible and hence the schedule is inconsistent.*)

If a schedule is verified to be inconsistent then it is repaired. The repair is done by planning from scratch with an existing \textit{xplan} and a \(\Delta\) combining the existing preferential, committed (i.e., the executed plan objects) and formal constraints.

The current CDSP model does not address all contingencies. It allows revisions of plans which do not affect plan object supports which are already committed. Also, it searches for a new schedule if for some reason, some plan objects from executed plans are non-countable (such as failing units in degree plans). Incorporating changes in existing plans due to changes in support requirements for plan objects which have their previous requirements committed is a particularly difficult task and requires temporal reasoning. We make a simplifying assumption in the current model that \textit{if there are changes in supports for plan objects in some plan during the course of its execution then they will not affect supports already committed}. For example, in degree planning, suppose in 1990, the execution of a plan \textit{plan}_{id} was started. Also, suppose \textit{plan}_{id} had the unit Comp2011 and (Comp1002 and (Sum-Wts Mathematics \(\geq\) 2)) as its prerequisites. If in 1990, Comp1001 had been undertaken and in 1991 the prerequisites
of Comp2011 changed to (Comp1001 and (Sum-Wts Mathematics ≥ 2)), we cannot disregard Comp1002 and include Comp1001 for the prerequisite for Comp2011.

Reasoning with changing formal constraints during plan execution is an interesting concern in plan evaluation, verification and repair. It is partially addressed in the current model and is suggested as an extension to the current research in Chapter 9.

6.6.4 Nogoods Database

The sources of arising inconsistencies during search in the developed problem solvers are represented globally in a nogoods database and exploited in future search by directing the search away from the selections that are already known to lead to failure. It directs selectivity of plan objects and supports incrementality in problem solving.

In each state, the cumulative structure captures the aggregation of variable assignments and hence, forms the searched structure in the context. On each state expansion, any inconsistent cumulative structures are detected and sets of plan objects contained by them are cached in the nogoods database. None of the constituent sets are supersets of other sets in the nogoods database. Every set in the database represents an environment that is known to fail and consequently avoided in future search. This failure is avoided by checking the extended cumulative structure \((CS' = S_i.CS \oplus css)\) against the nogoods database. If \(CS'\) contains a set of plan objects which is a superset of any of the sets in the nogoods database then \(CS'\) is declared inconsistent. By this checking, rediscovery of inconsistencies and futile traversal of search space are avoided.

The nogoods database is built up as inconsistencies arise in search. Initially, the plan object sets formed from the specifications of any Type 1 incompatibility constraints are placed in the nogoods database. That is, given \(C: \text{INCOMP} <\text{plan-object}> <\text{plan-objects}>\), nogoods = \{\{(Cons-labels \{C.plan-object\} \cup C.plan-objects)\}\. Presence of all the objects in \(\{C.plan-object\} \cup C.plan-objects\) in a cumulative structure would render the structure inconsistent. The size of the nogoods database is dependent on how informed the selection of plan objects and their support structures are. For more informed decisions, the nogoods database is small. For example, if the learned “islands” from COBWEB are not used as heuristics then the choices would be less informed and consequently, chances of getting a large number of nogoods during search would be higher.
6.7. Model-Search Formulation

6.6.5 Type-A Inconsistency

In the IPS machinery, any inconsistencies are found as soon as possible during search. Initially, as preferential constraints are added by the user, consistency (called Type-A inconsistency) between constraints in $\Delta$ are checked pairwise. If an inconsistency is detected, the user is asked to resolve it by retracting one of the culprits. Some examples of Type-A inconsistencies from the degree planning domain are:

- PLAN-TIME 3 years; PLAN-TIME 2 years,
- TAKE-IN comp3012 session1; OBJ-AVAIL comp3012 session2,
- TAKE-BEFORE comp2012 comp1012; PREQ comp2012 comp1012, and
- NOT-TAKE-OBJ-FROM chemistry; INCLUDE chem3012

6.7 Model-Search Formulation

6.7.1 Model-Search Problem

A CDSP problem is characterised as a Model-Search Problem

$$P = <\text{Plan-Objs}, \Delta>,$$

where Plan-Objs is a set of plan objects and $\Delta$ is a set of CL constraints ($\Delta = \text{Form-Cons} \cup \text{Pref-Cons}$). These constraints are Boolean expressions and belong to various groups: inclusion, exclusion, support, incompatibility, temporal, knapsack and committed constraints (as characterised in Chapter 3). It is a special case of the general CDSP problem stated in Chapter 3. In this formulation, the Form-Cons and Pref-Cons are not differentiated and all the constraints are treated as non-defeasible.

6.7.2 Model-Search Solution

A plan (or solution) to a Model-Search problem is an assignment $\rho = <B_1, B_2, \ldots, B_i, \ldots, B_n>$ such that all the constraints in $\Delta$ are satisfied. Recalling from Chapter 3, in an assignment, $B_i$s only contain plan objects.

6.7.3 Model-Search Solution Procedure

Given a Model-Search CDSP problem $P = <\text{Plan-Objs}, \Delta>$, the problem solving task is to search for a solution in an efficient way. We develop and present a solution procedure (called Model-Search) for solving Model-Search problems. The relation model-search $(P, S)$ relates Model-Search problems to Model-Search solutions.
The solution procedure developed for solving Model-Search problems is state-based tree-search. The search is characterised by the 5-tuple:

\[ \langle \Delta, S_0, \Gamma, GP, K \rangle. \]

A general state is given by \( S_i \): \( \langle AC, C, EX, X, CS, PC, schedule \rangle \) (the parameters are defined on page 62); \( S_0 \) is the initial state; \( \Gamma \) is a set of state transformation operators—these are the constraint satisfier and heuristic-based choice operators (as defined in Chapter 5), \( GP \) is the goal predicate, and \( K \) is the constraint-select operator which chooses constraints for application.

An inclusion constraint satisfier operator generates a set of successor states from a state and hence, gives non-determinism to search. A choice operator uses heuristics to select a single state from the resulting set of states.

The goal predicate \( GP \) takes a state \( S_i \) and evaluates all the constraints in \( \Delta \cup S_i.AC \) on \( S_i.schedule \). If all the constraints evaluate to true and \( S_i.schedule \) is fully elaborated then \( S_i \) forms the solution state \( S_g \). A fully elaborated schedule is a sequence of bins: \( <B_1, B_2, \ldots, B_i, \ldots, B_n> \) in which all \( B_i \)s contain plan objects only (i.e., a fully elaborated schedule is an assignment). Conversely, a schedule is partially elaborated when a bin \( B_i \) contains constraints in addition to plan objects.

![Figure 6.4: Search Space from Model-Search.](image)

Figure 6.4 illustrates the search space generated by Model-Search. From the initial state \( S_0 \), an inclusion constraint \( C_0 \) is chosen for application using the constraint-select algorithm. This application results in multiple successor states of which one is selected (in this case, \( S_{12} \)) through choice operators. The next best inclusion constraint is chosen
by constraint-select with respect to $S_{12}$. The chosen constraint $C_1$ is then applied. In the event of discovery of a failure along a path, control is chronologically backtracked revising earlier assignments. Each state is evaluated by $GP$ and the search terminates as soon as the goal state is found.

A solution is “fine-tuned” through user-addition of preferential constraints at runtime. This is achieved by the relation minimal-change $(P, S_1, C, S_2)$ where $P$ is the existing Model-Search problem, $S_1$ is the found Model-Search solution of the problem, $C$ is the run-time constraint, and $S_2$ is the minimally revised solution as induced by $C$ on $S_1$. If, upon an exhaustive search, a solution is not found then $\Delta$ is declared inconsistent.

All constraints are added and retracted through a meta-language. The minimal revision constraints are added at run-time whereas the others are added prior to invoking the search engine. Each change in constraints leads to a new CDSP problem and the new problem gets solved anew by Model-Search.

## 6.8 Description of Model-Search Procedure

An algorithm for the Model-Search ontology of CDSP problem solving is given in the next section. For now, a few informal comments may help clarify the procedure.

Model-Search is a dynamic constraint satisfaction approach based on a tree-search based algorithm for solving Model-Search problems. It takes a batch of constraints $\Delta$ and some existing plan (possibly, partially executed) and finds, in an efficient manner, an interpretation of the constraints that satisfies all the constraints in $\Delta$ (or model of $\Delta$). By adding preferential constraints and browsing their consequences, and retracting preferential constraints, Model-Search allows the user to “home in” on a plan with which he/she is satisfied. Model-Search treats each new batch of constraints as a new problem and searches for a solution anew for each new batch. It is an advisory approach in that it finds a solution based on some $\Delta$ and if the user is not satisfied then further constraints are added to $\Delta$ resulting in a new set $\Delta'$. Model-Search then finds a model for $\Delta'$. If, upon an exhaustive search, a model cannot be found then the Model-Search engine draws upon the experiential knowledge and suggests to the user which preferential constraints to retract or relax in order to get a model. This cycle, as demonstrated in Figure 6.5, continues until the user is satisfied with a plan.

Model-Search’s features include support for interactive browsing of existing solutions, queries on justifications for presence of plan objects in solutions, database queries, and user-addition and retraction of preferential constraints. Type-A inconsistencies
arising during search are also detected and identified to the user and resolved through help from the user. Model-Search keeps a trace of the problem solving activity from which it explains how a solution is found.

Model-Search employs a chronological backtracking search with an informed selection of constraints for application in search and an informed selection of values for the satisfaction of the constraints. Constraints get chosen dynamically and for each of them a variable and its associated domain are dynamically created. The properties of CDSP problems allow constraints to be differentiated into different functional categories of constraints which are expediently used in different stages of the search. Inconsistencies are avoided, if possible, in search by making informed selections for constraint satisfactions using available information. A COBWEB cluster and a consequent set of all the plan objects in existing CS provide useful heuristics in focusing search. This helps in guiding search away from potential inconsistencies. If, however, inconsistencies do arise, then they are cached in a nogoods database and any future search ensures that these inconsistencies are not re-discovered.

6.9 Model-Search Algorithm

6.9.1 A Descriptive Account

Here, we describe the Model-Search algorithm.

The state-space search model forms the basis for the constraint satisfaction engine
in Model-Search problem solving. Our task is to enhance and adapt the paradigm to incorporate heuristics, constraint-directed search, use of arising inconsistencies in future direction of search, and run-time plan revision.

Given an initial state $S_0$, a set of global constraints $\Delta$, a set of plan objects, a plan database, an executed plan database, and previous case clusters (as drawn by COBWEB), if available, the objective of the Model-Search problem solver is to search for a goal state $S_g$ using constraints and heuristics, such that $S_g.schedule$ is fully elaborate and satisfies all constraints in $\Delta \cup S_g.AC$. (Note that $\Delta$ is a global set of constraints, and $AC$ in a state contains a set of implicit or non-ground constraints which accumulate from support structures in search up to that state.)

In Model-Search, an initial state $S_0$, a set of (preferential and formal) constraints $\Delta$, a learned structure (as derived from CDSP-COBWEB) $LStruct$, and a database of objects capturing information from previous cases $Case-Info$ are globally defined. Prior to search, given the identifier for the plan being searched, the plan and executed plan databases are searched for any previously developed plan and any executed plan respectively. If an executed plan $xplan$ already exists for the plan identifier $id$ under consideration then $xplan$ is evaluated against the set of formal constraints from $\Delta$. If it satisfies all the constraints then problem solving stops with the $xplan$ being declared successful. Otherwise, $xplan$ and its corresponding plan are verified with the existing set of formal constraints. If they are found to be inconsistent then search gets invoked on the cumulative $CS$ as retrieved from the plan database and $\Delta$. Otherwise, if they are found to be consistent then the problem solving stops unless the user wants to revise the existing set of preferential constraints and search for a new plan.

Next, all the exclusion constraints from $\Delta$ are applied. They are used to filter the domain plan objects with the plan objects specified for exclusion from plans.

In search, since each of the cases in the $Case-Info$ database are admissible (i.e., consistent with formal constraints), search is performed on the database for one case which is consistent with all the preferential constraints. If one such case is found from the “shelf” then the problem solving terminates with it. Otherwise, an initial state is established and search begins from scratch for a new plan.

During search, given a state $S_i$, an inclusion constraint $C$ is chosen from $S_i.AC \cup \Delta$ using the constraint-select algorithm. This selection is dynamically made at the state $S_i$ with respect to information contained in $S_i$ and on the heuristic-based selection criteria given in §6.5. Before applying $C$ to $S_i$, a variable is dynamically created for the constraint and its associated domain is dynamically determined with respect

---

6.9. Model-Search Algorithm

139
to the existing state $S_i$. $C$ is then applied to $S_i$. The application requires search for a set of plan objects $EX$ (called expansion of $C$), a conjunctive support of which, when added to $CS$ would result in another $CS$ that is schedulable and consistent with $\Delta \cup S_i.AC$. A successful application of $C$ to $S_i$ results in $S_{i+1}$. This includes the case when $C$ is already satisfied by $S_i$. In this case, $S_{i+1}$ is generated by inheriting information from $S_i$ with $S_{i+1}.PC$ updated with $C$. If, however, $C$ cannot be successfully applied then backtracking is invoked. In backtracking, the chronologically previous variable assignment is undone and another alternative is assigned. Prior to each variable assignment, the extension of the existing $CS$ with the chosen conjunctive support structure is checked against nogoods. If an exhaustive search does not result in a solution then $\Delta$ is declared inconsistent. Model-Search then gives advice to the user on possible constraints to add which it draws from experiential knowledge and based on the existing preferential constraints. It finds a case from the case database that is consistent with a maximal number of preferential constraints and suggests it to the user. A solution path $SP$ is maintained for each solution for the purposes of explanations.

The most important difference between the search strategy employed in Model-Search and the existing heuristic-guided search strategies is that in Model-Search, the goal state is not known prior to the start of search. Hence, heuristic-based distance estimates from a transition state to the goal state cannot be used in guiding the search. Whereas, in traditional heuristic-guided search strategies, such as $A^*$ (Korf surveys recent results in search in [90]), the goal state is sufficiently specified so that a cost metric is effectively used in directing search towards it. The objective of these strategies is to find a solution path with minimal total cost. In Model-Search, however, since new constraints emerge during search, the specification of goal state changes dynamically as search progresses. Consequently, Model-Search does not use a heuristic estimate. It heuristically selects an inclusion constraint for application and expands the existing state by heuristically finding plan objects and their supports to the cumulative structure in the existing state such that the selected constraint evaluates to true in the new state. The heuristics used in selecting the values for satisfying the different kinds of constraints at each state have been discussed in Chapter 5 (see §5.3.2.2, page 95).

### 6.9.2 Algorithm

The Model-Search algorithm embodies the following important functions for constraint selections, constraint-driven state expansions and constraint evaluations.
• evaluate-plan: \( XPLANS \to P \text{ CONST} \cup \{\text{pass}\} \).

An executed plan is evaluated against an existing set of formal constraints. \( \text{pass} \) is returned if all the formal constraints are satisfied. Otherwise, a set of violated constraints is returned.

• verify-plan: \( Schedules \times XPLANS \to \text{Boolean} \).

An existing partially executed schedule is evaluated against an existing set of formal and preferential constraints. True is returned if the schedule is consistent with all the constraints. Otherwise, false is returned.

• constraint-select: \( P \text{ CONST} \times CSS \to \text{CONST} \cup \{\text{nil}\} \).

Constraints are automatically selected for application by the above function.

• apply-constraint: \( \text{CONST} \times S \to S \).

After an inclusion constraint selection, \( \text{apply-constraint} \) is invoked on the constraint and an existing state resulting in a new state. This function embodies heuristics and choice operators to determine an optimal expansion from usually multiple expansion possibilities. It also uses nogoods.

• goal-state?: \( S \to \text{Boolean} \).

On generation of each new state by the search engine, the goal conditions are tested by the function \( \text{goal-state?} \) on the state. If all the constraints in the state evaluate to true and the schedule contained in it is fully elaborated then it is a goal state and the search terminates with it.

• user-satisfied?: \( Schedules \to \text{Boolean} \).

The problem solver stops as soon as the user is satisfied with a schedule.

Below, we give an algorithm for Model-Search. It is divided into two components: Model-Search0 and Model-Search1. Model-Search0 reads and sets up the data, acquires user constraints and performs pre-processing of the constraints. Then, it invokes Model-Search1 which is the informed backtrack tree-search engine for constraint satisfaction. Model-Search1 is also used by the IPlan procedure. Note in the algorithms, we have introduced another parameter in the states called \( \text{digest} \). It facilitates computations on backtracking by recording already considered choices at each node. In the following, first, the variables used in both the components are given.
Model-Search

Variables

Plan-objs: A set of all plan objects.

WS: A working set of plan objects.

Form-Cons: A set of formal constraints.

Pref-Cons: A set of preferential constraints.

id: A unique plan identifier.

C: A constraint.

CS: A cumulative structure.

schedule: A schedule—reverse of a plan.

xplan: An executed plan.

nogoods: A set of nogoods.

change-cons: A run-time constraint.

S_0: Initial state.

S_g: Goal state.

S_i.AC: Accumulated constraints in state S_i.

S_i.C: Constraint applied to S_i.

S_i.digest: Choices already considered to satisfy S_i.C.

S_i.EX: Expansion set of S_i.C.

S_i.X: A conjunctive support structure for S_i.EX.

S_i.CS: Cumulative structure in S_i.

S_i.PC: Sequence of constraints applied upto S_i.

S_i.schedule: One schedule of S_i.CS.

LStruct: A COBWEB cluster.

Case-Info: A database of case objects.

c-node: A state node in tree-search.

n-node: A state node in tree-search.

OPEN: A sequence of state nodes.
**Inc-Cons:** A set of inclusion constraints.

**SP:** A solution path—a sequence of state nodes.

---

**Model-Search0**

\[ \text{model-search0: } () \rightarrow \text{Schedules} \cup \text{XPLANS} \cup \{\text{fail}\} \]

**{Initialisation}**

Input \( id, \text{Pref-Cons}, \text{Form-Cons}, CS, xplan, schedule. \)

(*\text{Pref-Cons, CS, xplan, schedule may already exist for } id.*

\[ \text{user-satisfied } \leftarrow \text{false} \]

\[ \text{change-cons } \leftarrow \text{nil} \]

\[ \text{nogoods } \leftarrow (\text{get-plan-obj-sets (get-type1-incomp-cons Form-Cons)}) \]

(*\text{nogoods from Type 1 incompatibility constraints.}*

**{Set-Up Data}**

\[ \text{if } 3 \text{ schedule for } id \text{ then} \]

\[ \text{if (evaluate-plan } xplan) = \text{pass} \text{ then (return } xplan) \text{ fi} \]

\[ \text{if (not (verify-plan } schedule xplan) \text{) then } schedule \leftarrow \text{nil} \]

(*\text{schedule is inconsistent with Form-Cons.}*

\[ \text{else } S_y.\text{schedule } \leftarrow \text{schedule} \]

(*\text{schedule is consistent but further revision may be required.}*

\[ \text{fi} \]

\[ \text{else} \]

\[ \text{schedule } \leftarrow \text{nil} \]

\[ \text{Pref-Cons } \leftarrow \text{nil} \]

\[ CS \leftarrow \text{nil} \]

\[ xplan \leftarrow \text{nil} \]

\[ S_y.\text{schedule } \leftarrow \text{nil} \]

\[ \text{fi} \]

\[ \Delta \leftarrow \text{Form-Cons } \cup \text{Pref-Cons} \]

(*\text{All initial constraints.}*

**{Main Loop}**

\[ \text{until user-satisfied do} \]

\[ \text{change-cons } \leftarrow (\text{ask-user-for-change-cons}) \]

\[ \text{if change-cons } \neq \text{nil then} \]

\[ \text{schedule } \leftarrow (\text{process-change-cons change-cons schedule CS}) \]

(*\text{change-cons induces minimal changes in schedule.}*)
else

    Pref-Cons ← (ask-user-for-constraints schedule Pref-Cons)
    (*Any preferential constraints are acquired from user.*)
    \( \Delta \leftarrow \Delta \cup \text{Pref-Cons} \)

    WS ← (apply-exclusion-constraints (get-exclusion-cons \( \Delta \)) Plan-Obj)
    (*Process exclusion constraints.*)

    schedule ← (match-retrieve-model Pref-Cons Case-Info)
    (*A schedule consistent with Pref-Cons is searched from case database.*)

fi

if schedule = nil then
    \( S_g \leftarrow \) (model-search1 schedule CS)
    if \( S_g = \text{fail} \) then
        (update-browser (give-advice \( \Delta \) LStruct Case-Info))
        elseif (user-satisfied? \( S_g.\text{schedule} \)) then
            (save-plan-db id Pref-Cons \( S_g.\text{schedule} \) CS)
            user-satisfied ← true
        fi
    fi
endif

(answer-user-queries)

(return schedule)

---

Model-Search1

\textit{model-search1: Schedules \times CSS \rightarrow S \cup \{\text{fail}\}}

\textbf{Input:} sched:Schedules, CS:CSS

\{Set-Up Initial State and Solution Path\}

\( S_0.\text{AC} \leftarrow \) (get-cons CS)

\( S_0.C \leftarrow \text{nil} \)

\( S_0.EX \leftarrow \text{nil} \)

\( S_0.X \leftarrow \text{nil} \)

\( S_0.PC \leftarrow () \)

\( S_0.\text{schedule} \leftarrow \text{sched} \)

\( S_0.\text{digest} \leftarrow \{} \)

\( SP = \{} \) (*\( SP \) is a set of (predecessor successor) tuples.*)
6.9. Model-Search Algorithm

{Backtrack Search Engine}

if (goal-state? \( S_0 \)) then (return \( S_0 \)) fi

(push \( S_0 \) OPEN)  (*OPEN is a stack.*)

Label: if \( OPEN = \text{null} \) then (return fail) fi

c-node \( \leftarrow \) (first-elt OPEN)

if \( \exists c\text{-node}.C \) then

\( \text{Inc-Cons} \leftarrow \text{(get-inclusion-cons} \Delta c\text{-node.AC} c\text{-node.PC)} \)

\( C \leftarrow \text{(constraint-select Inc-Cons c\text{-node.CS)} \}

\( c\text{-node}.C \leftarrow C \)

else \( C \leftarrow c\text{-node}.C \)

fi

if \( C = \text{nil} \) then (remove c\text{-node OPEN}) and go to Label fi

(*All inclusion constraints are satisfied but goal-node is not reached.*)

n-node \( \leftarrow \text{(apply-constraint} C \text{c\text{-node)} \}

(*C is applied by selecting a conjunctive support structure which is not already
contained in c\text{-node.digest}.*)

if n-node = fail then

(*Expansion of c\text{-node not possible.*)

(remove c\text{-node OPEN}) (*c\text{-node removed from the stack.*)

(remove-tuples-sp c\text{-node}) (*Solution path is updated.*)

go to Label

elseif (goal-state? n-node) then

(return n-node)

else

if n-node.X \( \neq \) nil then

\( c\text{-node.digest} \leftarrow c\text{-node.digest} \cup \{n\text{-node.X}\} \)

(*Choices already made at each node are recorded in the node.*)

fi

(push n-node OPEN)

\( SP \leftarrow SP \cup \{(c\text{-node n-node)}\} \)

go to Label

fi
The Model-Search algorithm dynamically selects constraints for application through the constraint selection algorithm \textit{constraint-select} as given in §6.5.2. The selected constraint is applied through \textit{apply-constraint} which forms the underlying generator of the search space. \textit{apply-constraint} applies the appropriate constraint satisfier operators as discussed in Chapter 5. Each application requires a heuristic-guided choice of a conjunctive support structure for an expansion for $C$, which, when added to an existing $CS$ results in a $CS$ that is consistent with all constraints in $\Delta$ and schedulable. The constraint variables, $Xs$, are assigned conjunctive support structure values depending on previous variable assignments through constraint propagation. (Constraint application is described in the previous chapter.) Model-Search essentially relies on constraints as the underlying control in search. The constraints specify dimensions to be searched.

### 6.10 Explanations and Revisions

The user is allowed to seek explanations from the system at run-time. WHY <\textit{plan-object}> derives the justification for the presence of \textit{plan-object} in the existing schedule from the solution path and $CS$ structure. GIVE-TRACE gives the trace of the solution path displaying constraint variable assignments in each state.

In Model-Search, each change in the constraints is treated as a new problem and the problem is solved anew. Thus, the user is allowed to explore solution possibilities by adding and retracting constraints. At run-time, Model-Search allows the user to minimally revise a found solution through run-time constraints.

### 6.11 Analysis of Model-Search

Model-Search is a dynamic constraint satisfaction approach. For each new batch of constraints, a solution is found if one exists. Pre-processing of some constraints contribute to making search efficient. This is performed prior to running the search. The search is informed in that domain knowledge and already discovered failures guide it.

The major strength of Model-Search is its ability to use constraints in focusing search. The inclusion constraints are used as generators. They remain unapplied unless their applications are forced. The constraint-select algorithm decides on which constraint to apply at a given point in search. The nogoods database prevents re-discovery of those points in search that are known to fail and hence, avoids the thrashing be-
6.11. Analysis of Model-Search

haviour in search. Thus, only promising solution paths are explored and consequently, backtracking is minimised.

The search is complete i.e., for a given set of constraints, Model-Search finds a solution if there is one. During search, any arising inconsistencies are detected as soon as possible and the appropriate solution paths are excluded from further exploration. This is based on the monotonic nature of the search that if an inconsistency is detected on a solution path then no further exploration of the path will re-establish consistency. That is, if a sequence of constraints \(<C_1, C_2, \ldots, C_j>\) is known to be inconsistent then no extended sequence \(<C_1, C_2, \ldots, C_j, C_{j+1}>\) can possibly be consistent; hence, by induction no extended sequence \(<C_1, C_2, \ldots, C_j, C_{j+1}, \ldots, C_n>\) can be consistent.

A Model-Search solution is sound in that a solution, if one is found, does not violate any of the constraints.

Since any change in constraints renders an existing problem a new Model-Search problem, search for solutions from scratch for each new problem is computationally an expensive task. This does not allow flexibility in exploring the solution possibilities which is an essential component of CDSP problem solving.

If a solution for a given set of constraints cannot be found, the current Model-Search model does not find a solution that satisfies all the formal constraints and a maximal set of the preferential constraints. The defeasible nature of preferential constraints may allow search for such solutions. However, this feature has not been realised in the current Model-Search model but the properties of user-directed and incremental problem solving as captured in the IPLAN formulation realises this feature per se.

Although Model-Search embodies a new approach in solving constraint satisfaction problems of the CDSP kind, it is cumbersome in user-directed and incremental development of solutions as required by CDSP problems. We need a solution procedure that extends Model-Search to facilitate user interaction and incrementality in the solution process.
Chapter 7

Incremental Planning: IPLAN

7.1 Introduction

The Model-Search approach for constraint satisfaction in the CDSP class of problems requires that a batch of constraints is known prior to search. It does not allow constraints to be added or retracted during search. Since an essential aspect of CDSP problems is the interactive human dimension in guiding problem solving, it is particularly desirable that the problem solver facilitates dynamic addition and retraction of preferential constraints during problem solving. User preferences allow exploration of possibilities by inducing changes and revision as a result of any arising contingency. There needs to be a flexible mechanism that allows such revisions. Also, it is essential that problem solving is incremental in which previously explored information is made available to future problem solving. Thus, unlike Model-Search, each dynamic addition of constraints has to be treated as a change to the initially defined problem and not treated as a new problem. The incrementality allows satisfaction of all preferential constraints defined up to each point in problem solving and the question of maximally satisfying the preferential constraints does not arise. These features are missing in the Model-Search formulation and motivate its extension.

In this chapter, we extend the Model-Search formulation to capture the above dynamic properties of CDSP problem solving. CDSP problems are formulated as IPLAN (standing for Incremental PLANning) problems. IPLAN problems and their solutions are characterised, and an IPLAN solution procedure is developed. An ATMS-type computational tool is suited to the properties of incremental problem solving. The truth maintenance system (TMS) technology and in particular, the ATMS, is briefly reviewed. The advantages to CDSP problem solving on coupling it with a de Kleer type ATMS [41] are identified. Inadequacies in using just the basic ATMS in IPLAN
problem solving is argued. A single-context ATMS (called CDSP-ATMS) coupled with dependency-directed backtracking (DDB) control is found to be suitable in IPLAN problem solving. Finally, an interactive and incremental algorithm for IPLAN problems is presented in terms of the CDSP-ATMS-DDB combination.

7.2 IPLAN Formulation

Having given a formulation of a CDSP problem as a Model-Search problem, and a characterisation and description of its solution and solution procedure respectively in Chapter 6 (see page 135), we now extend the formulation such that it models the user behavior. The motivation for this model is the need for incremental and interactive development of solutions as required in CDSP problem solving. Rather than automatically searching for a solution anew for each batch of constraints, user-guided incremental development of solutions through a sequence of partial solutions is required.

7.2.1 IPLAN Problem

A CDSP problem is characterised as an IPLAN problem

\[ P = < \text{Plan-Objs}, \text{Form-Cons}, \text{Cons-Seq}>, \]

where Cons-Seq is a finite sequence of preferential constraints, which may be empty, and the other definitions are as given in the Model-Search problem characterisation earlier on page 135.

7.2.2 IPLAN Solution

Given an IPLAN problem \( P = < \text{Plan-Objs}, \text{Form-Cons}, \text{Cons-Seq}>, \) where Cons-Seq = \( <C_1, C_2, \ldots, C_i, \ldots C_k>, \) a solution sequence SSeq of \( P \) is \( <\Delta_1, \varphi_1>, <\Delta_2, \varphi_2>, \ldots, <\Delta_i, \varphi_i>, \ldots, <\Delta_k, \varphi_k>> \) where \( \Delta_1 = \text{Form-Cons} \cup \{C_1\}, \) each \( \Delta_i \) is \( \Delta_{i-1} \cup \{C_i\}, \) and each element \( <\Delta_i, \varphi_i> \) is a partial solution.

In a partial solution, \( \Delta_i = \text{Form-Cons} \cup \{C_1, C_2, \ldots, C_i\}, \) and \( \varphi_i = <B_1, B_2, \ldots, B_j, \ldots, B_n> \) where the \( B_j \)s contain elements from \( \text{Plan-Objs} \cup \text{CONST} \) (CONST is a set of sentences in CL). \( \varphi_i \) contains the consequences of constraints that are forced, and is consistent with all the constraints in \( \Delta_i \) and any constraints contained in \( \varphi_i \). It may be partially elaborated. A \( \varphi_i \) is partially elaborated when a bin \( B_j \) in \( \varphi_i \) contains constraints in addition to plan objects. These constraints restrict addition of plan objects in \( B_j \) and bins below it in future steps. Conversely, when there are no constraints in the \( B_j \)s (i.e., when the bins contain plan objects only), then \( \varphi_i \) is said to be fully elaborated. Each
Chapter 7. Incremental Planning: IPLAN

A partial solution for an IPLAN problem can be easily mapped to a Model-Search problem and solved using the Model-Search problem solver. Each of the partial solutions can be viewed as new Model-Search problems.

A \( \langle \Delta_k, q_k \rangle \) is an IPLAN solution iff \( q_k \) is fully elaborated and all the constraints in \( \Delta_k \) when evaluated on \( q_k \) result in \text{true}.

### 7.2.3 IPLAN Solution Procedure

Given an IPLAN problem \( P = < \text{Plan-Objs}, \text{Form-Cons}, \text{Cons-Seq} > \), the objective of the IPLAN problem solver is to incrementally develop a solution sequence using the user as an oracle in guiding the search. We develop and present a solution procedure for IPLAN problems (called IPLAN). The relation \( \text{iplan} (P, SSeq) \) relates IPLAN problems to their solution structures.

The partial solution at each point belongs to a spectrum of possibilities, \( \langle \Delta_1, q_1 \rangle, \langle \Delta_1, q_2 \rangle, \langle \Delta_1, q_3 \rangle, \ldots, \langle \Delta_i, q_i \rangle, \ldots, \langle \Delta_k, q_k \rangle \rangle \) where \( q_1 \) is empty, and \( q_k \) is fully elaborated and consistent with all the constraints in \( \Delta_i \); i.e., \( q_k \) is a Model-Search solution of \( \Delta_i \).

In the IPLAN solution scheme, the solution is incrementally and systematically developed through the user constraints. The user adds a constraint, browses the resulting partial solution, seeks advice (if needed), formulates a new constraint and continues this cycle until all user preferences are exhausted. Each consecutive addition of a constraint results in a tighter partial solution. Rather than finding a solution anew for each addition of new constraints, IPLAN retains a solution structure that captures all partial solutions in a sequence. This facilitates the incremental development of solutions.

In the incremental development of solutions, Model-Search solutions are not required although they can be found from partial solutions in situations where solutions are not achieved incrementally through preferential constraints.

Reasoning with constraints depends on whether, on addition of constraints

- **Case A**—the constraints are monotonically increasing, or
- **Case B**—the constraints are changing nonmonotonically.

### Case A

In the case of monotonically increasing constraints during problem solving, the constraints become tighter with time (i.e., there is a total ordering on the partial solutions). This is a useful and characteristic property of all constraint satisfaction problems. If
7.3. Truth Maintenance Technology

In search for solutions in AI problem solving, there are often competing alternatives from which choices are made. It is usually difficult to make the right choices all the time. When the choices made lead to failure they have to be revised. As we have seen in the previous chapter, standard backtracking and dependency-directed backtracking are two such search techniques which recover from failures in different ways. The main objective of any search technique is that futile traversals of search space are avoided. This is done either by making informed choices (called \textit{a priori pruning}) to avoid failure, an “intelligent” recovery from failure (called \textit{a posteriori pruning}), a mixture of the two, or maintenance of all feasible partial solutions to avoid backtracking search. The basic motivation for the development of truth maintenance systems \cite{41,49,102} (also called \textit{belief revision} or \textit{reason maintenance} systems) was to provide general solution to the search problem using computational mechanisms which effectively and efficiently
support revision in nonmonotonic reasoning\(^1\). These mechanisms track the effects of changes through data dependencies.

Figure 7.1: Inference Engine/TMS Interaction in a Reasoning System.

In general, a TMS acts as a house-keeping sub-system of a reasoning system (see Figure 7.1). It is designed to enhance the efficiency of problem solving by sharing information across regions in search space of solutions or partial solutions. This is made possible by caching results obtained in one region of a search space and making them available to other regions of search. This attempts to avoid re-computations. The problem solver draws inferences and passes them to the TMS and the TMS uses the structure of these inferences to organise and house-keep the beliefs of the problem solver. These inferences are recorded as justifications by the TMS, never re-computed, and made available to other regions in the search space. The TMS serves as a cache for all the inferences ever made—inferrnces once made, are not repeated, and inconsistencies once discovered, are avoided in the future. It allows the problem solver to make nonmonotonic inferences and ensures that the database is consistent [49]. At some point in problem solving, the problem solver acquires information from the TMS on what it believes based on the inference information it is supplied. The form of the interaction between the two sub-systems varies, but typically the TMS would provide information about the beliefs on which inconsistencies rest and sets of mutually consistent beliefs with which to work. The TMS has no access to the semantics of the problem solver behavior. The expressions passed by the problem solver are treated syntactically.

de Kleer, Forbus and McAllester [36] have identified five general reasons for incorporating a TMS in a problem solver. They are summarised below.

- Responsibility for Conclusions

A problem solver needs to explain what is responsible for its conclusions. This

\(^{1}\)Reasoning with constraint conflict and other changing conditions during problem solving is generally captured by the notion of nonmonotonic reasoning and formalised by several logics developed in AI. Examples are Default Logic [129], Nonmonotonic Logic [103], Circumscription [100] and Autoepistemic Logic [89].
7.3. Truth Maintenance Technology

helps a user in exploring changes in events of contingencies.

- **Recovery from Inconsistencies**
  By keeping justifications it is possible to easily track what causes inconsistencies. From this explanation, the culprits can be easily identified and appropriate decisions revisited to recover from inconsistencies.

- **Caching of Inferences**
  Caching of inferences is a powerful mechanism. It avoids futile traversal of search space. Inferences already made are not re-computed and rediscovery of any failures in the past are avoided in the future.

- **Guided Backtracking**
  The recording of justifications allows easy detection of choices on which any arising inconsistencies depend and the backtracker backtracks to the most recent choices contributing to the inconsistencies. This is *dependency-directed backtracking* (see Chapter 6, page 117 for a full description of dependency-directed backtracking) and is enabled by TMSs.

- **Default Reasoning**
  TMSs allow making conclusions based on insufficient information. By default, conclusions are made and on emergence of contrary evidence during problem solving, TMSs allow a graceful retraction of these inferences.

There are several different kinds of TMSs. They are, in order of invention, Justification-Based TMS (JTMS) [49], Logic-Based TMS (LTMS) [98], Assumption-Based TMS (ATMS) [41]. Each kind is appropriate for different problem solving tasks. JTMSs and LTMSs force the problem solver to reason in one context at a time. They are single context TMSs in which there is always a single contradiction-free database (called a context). This is a fundamental drawback as many real-world problem solving tasks require reasoning with multiple contexts\(^2\) concurrently. This drawback motivated the design of ATMS as prompted from applications in qualitative reasoning [33] and circuit diagnosis [34]. Since the invention, ATMSs have become an increasingly popular tool in AI problem solvers.

The choice of which TMS to use in practice depends largely on the kinds of supports that are required from the problem solving systems. In situations where there

\(^2\)In a single context TMS, only one contradiction-free set of data is communicated to the problem solver whereas in a multiple context TMS, several consistent sets of data—possibly, mutually inconsistent—may be available simultaneously.
are many solutions and only a few are required, a single context TMS is more appropriate as it focuses search by retaining dependencies that are relevant. It is unsuitable for applications where several partial solutions are to be considered and compared in the process of developing complete solutions. An assumption-based multiple context system supports such a feature by succinctly capturing the choices already made and allowing switching of contexts to compare solutions. It supports an incremental and exploratory (through hypothetical reasoning) problem solving. These features are desired in many real-world applications. To realise the potential of an ATMS in problems with combinatorial solution space, it is essential that problem solving is organised in a manner which allows restrictions on partial solution possibilities to promising possibilities during problem solving such that the number of possibilities is manageable. In interactive systems, this can be achieved through user-directed exploration and pruning of possibilities. This makes the use of multiple context ATMS in problem solving computationally efficient.

7.4 de Kleer's ATMS

CDSP problem solving involves reasoning with a variety of constraints and requires a novel use of de Kleer type ATMS. Here, we summarise the ATMS from [30,31,149]. The ATMS theory and its various extensions are explained in [29,30,34,35,41,149].

An ATMS-based problem solver consists of an inference engine coupled to an ATMS like in Figure 7.1. The task of the inference engine is to make inferences about the domain, and the task of the ATMS is to determine, given the inferences made by the inference engine, what is and what is not believed. Every datum which the ATMS reasons about is assigned an ATMS node which forms the basic structure on which the system depends. The inference engine designates a subset of the nodes to be assumptions. These nodes are presumed to be true unless there is evidence to the contrary. A set of assumptions is an environment and the set of all data derivable from the assumptions is the context of the environment. If a contradiction (i.e., false) is derivable from the assumptions, then the environment is inconsistent and the assumptions are taken to justify a distinguished node \( \bot \) which designates false. All inferences are recorded and communicated to the ATMS as justifications.

\[ x_1, \ldots, x_k \Rightarrow n. \]

Here, \( x_1, \ldots, x_k \) are antecedent nodes and \( n \) is the consequent node.

The task of the ATMS revolves around its ability to determine, for a given da-
7.4. de Kleer’s ATMS

tum, the assumptions under which it holds. In the de Kleer notation, the nodes are represented as follows:

\[ \gamma_{\text{datum}}: \langle \text{datum, label, justifications} \rangle \]

The datum is the problem solver datum with which it is associated. It is treated atomic by the ATMS. Each justification consists of the antecedents of one of the problem solver inferences used to support the datum. The label is computed by the ATMS. It caches a set of environments (i.e., a set of sets of assumptions) each element of which, if made together, allow the datum to be inferred. The ATMS does propositional reasoning over these nodes. Every node is a propositional symbol, and every justification is a Horn clause. An environment is a conjunction of propositional symbols.

A node \( n \) is said to hold in an environment \( E \) if \( n \) can be propositionally derived from the union of \( E \) with a set of clauses \( J \) corresponding to the justifications which have been communicated to the ATMS. An environment is inconsistent (called nogood) if \( \bot \) holds in it. A nogood is minimal if it contains no others as a subset.

An ATMS is incremental, receiving a constant stream of additional nodes, additional assumptions, additional justifications and various queries concerning the environments in which nodes hold. To facilitate answering these queries the ATMS maintains for each node \( n \) a set of environments \( \{E_1, \ldots, E_k\} \) (called the label) having the following four properties:

1. \( n \) holds in each \( E_i \) [Soundness].
2. \( E_i \) is not nogood [Consistency].
3. Every environment \( E \) in which \( n \) holds is a superset of some \( E_i \) [Completeness].
4. No \( E_i \) is a proper subset of any other [Minimality].

The information recorded in the ATMS facilitates an efficient search in two ways: incrementality and selectivity [138]. Recording justifications gives incrementality. It avoids re-discovery of sections of solution space already discovered and hence, the problem solver is allowed immediate access to inferences it has already made. Inferences made in one section of the search space is made available to other sections and therefore not wasted. The recording of nogoods allows selectivity whereby the problem solver selectively chooses candidates from choice points such that failures discovered in the past are not re-discovered in the future.

The algorithms for the creation and maintenance of an ATMS are given in [31,36]. The interfacing of an ATMS with a problem solver is discussed in [31].
7.5 IPLAN and ATMS

The motivation behind developing the IPLAN approach is to permit the user to browse transition solution states, and add preferential constraints interactively exploring and guiding search for one solution. IPLAN captures reasoning for dynamic constraint satisfaction. Its characteristic feature is incrementality through which any change in the constraints does not start the constraint satisfaction process anew but the previously processed information is used for further problem solving. This feature is summed up in Figure 7.2. It depicts increase in information content in problem solving as the problem solving progresses. IPLAN ameliorates Model-Search by incorporating this incrementality feature in problem solving and allowing more control to the user in interacting and controlling the problem solver at each stage of problem solving.

The coupling of an ATMS to a CDSP problem solver explains the increasingly justified belief that real-world tasks which are modeled by expert system technology need to be incremental and interactive. The reasoning engine must support shared hypothetical reasoning between a human and an expert system i.e., at any time during problem solving the system or the user can backtrack to an earlier decision and revise it. The system must facilitate acquisition of new knowledge from the user, and retraction of constraints and decisions with minimal backtracking. Also, it has to maximally exploit knowledge at its disposal in searching for a solution. An ATMS facilitates these
problem solving functionalities.

In the following subsections, we discuss IPS problem solving, discuss how a multiple-context ATMS might be used with it to get the desired functionality, discuss the drawbacks in using a multiple-context ATMS, motivate and develop the IPLAN approach in terms of the proposed CDSP-ATMS, and the dependency-directed backtracking scheme.

### 7.5.1 IPLAN Problem Solving

The IPLAN architecture conforms to the general CDSP problem solving architecture given in Chapter 3, on page 52.

In the IPLAN problem solving approach, the user plays an important role in interactively guiding and focusing the search. The user formulates constraints in CL and issues them to the problem solver until all the preferential constraints are exhausted or a solution is found. The preferential constraints are processed dynamically in the order they are received, respecting all the formal constraints. When all the preferential constraints are exhausted and a solution not fully developed then the backtrack engine of Model-Search, model-search1, is invoked on some existing IPLAN state to automatically extend it to a goal state i.e., model-search1 carries the existing IPLAN partial solution to completion. The existing partial solution satisfies all the preferential constraints and is consistent with all the formal constraints. The invocation of model-search1 applies to the existing state all inclusion constraints from the formal constraints in order from most constraining to least constraining (as discussed in Chapter 6). This application tests the existing solution against each of the formal constraints and takes action to satisfy them. If an exhaustive search does not result in a solution then the combined set of preferential and formal constraints is declared inconsistent.

During the interaction with IPLAN, the user formulates his/her own preferential constraints with respect to the existing problem solving state. If advice is sought from the system, the advisor component of the system retrieves information from the COBWEB clusters based on the existing partial solution and a set of unapplied inclusion constraints from formal constraints, and displays to the user a set of possible constraints as advice.

The various kinds of constraints in CDSP problems are effectively used in the IPLAN problem solver. We have seen the advantages in their difference and use in different stages of Model-Search problem solving in the previous chapter. In IPLAN too, as a new constraint is received during problem solving, its category is determined and an
appropriate action taken contingent upon which class the constraint belongs. If it is an inclusion constraint then the existing state is expanded such that the new constraint is satisfied in the new state. If it is a temporal constraint then the solution in the existing state is re-scheduled to ensure that the new constraint is satisfied. If it is an exclusion constraint and if the existing solution is inconsistent with it then the current solution is revised to make it consistent with the constraint. Also, it is ensured that any future selections of plan objects are consistent with the exclusion constraint. How these constraints are dynamically processed is the subject of the control structure in IPLAN. The IPLAN algorithm is an extension of the Model-Search algorithm whereby it couples the CDSP-ATMS to house-keep information in each state of the search. This architecture facilitates browsing of information in each state by the user and permits user control in decision making at each stage in IPLAN problem solving.

7.5.2 Inadequacy of ATMS in IPLAN Problem Solving

An ATMS is well-suited for managing the reasoning process in which several partial solutions are to be considered and compared, and concurrently and progressively developed in pursuit of multiple solutions. Hence, in IPLAN problem solving, support of an ATMS is required. Other TMSs do not support such incremental problem solving features.

As each inclusion constraint is added at some point in the problem solving process, there are usually multiple ways of satisfying it. An ATMS can capture these multiple possibilities as multiple contexts and succinctly capture and maintain justification structures. It enables the user to analyse incrementally the impacts of changes on solutions when new constraints are added. It also facilitates exploration of alternative solutions through what-if analysis. It is desirable to have these features supported in the reasoning system.

The major drawback in using a multiple-context ATMS in managing control in IPLAN problem solving is that it is computationally very expensive to develop and maintain all partial solutions during the course of problem solving. Even maintaining a small number of multiple partial solutions is computationally expensive. This is explained using the simple search space illustration of Figure 7.3. The initial state in problem solving is given by $S_0$. Application of an inclusion constraint to an existing problem solving state usually results in multiple successor states. From the figure, $S_0$ expands to $S_1 \ldots S_n$; $S_1$ expands to $S_{11} \ldots S_{1k}$; $S_2$ expands to $S_{21} \ldots S_{2l}$ and so on, where $k, l, m, n$ and $p$ are positive integers which may be large but are finite. The
Constraint Addition

\[
C_1 \\
S_0 \\
C_2 \\
S_1 \\
S_2 \\
S_3 \\
S_n \\
S_{11} \\
S_{12} \\
S_{1k} \\
S_{21} \\
S_{22} \\
S_{2l} \\
S_{n1} \\
S_{n2} \\
S_{nm} \\
S_{21} \\
S_{212} \\
S_{21r} \\
S_{21y} \\

\]

Figure 7.3: A Search Space in CDSP Problem Solving.

contents of the successor states depend on information in the existing state. \( S_2 \ldots S_n \) are derived from the application of \( C_1 \) to \( S_0 \).

Like in Model-Search, the application involves a distance calculation between \( CS \) in the existing state and the goal state in which the added constraint is satisfied. A set of plan objects is heuristically selected based on this distance and a conjunctive support structure which results in a consistent extension of the existing \( CS \) in the updated state. That is, each of the states in Figure 7.3 contain a \( CS \) which is consistent with respect to all the formal constraints and any constraints accumulated upto each state in their respective path traversals.

Unlike Model-Search, the increments to satisfy an inclusion constraint cannot be easily reasoned with and incorporated in the solutions because the distances may be different for different states, and different inclusion constraints may be optimal for application to each of the different states. Also, for each inclusion constraint application, the local constraints, including schedulability, need to be checked in each context. Whereas Model-Search traverses a single path of the search space in a depth-first manner avoiding any nogoods and backtracks if inconsistency arises, in a multiple context scenario, all the states at each level are explored concurrently, in a breadth-first manner. The cumulative structures contained in these states define multiple partial solutions. Application of an inclusion constraint to each of these states may produce further multiple successor states giving rise to even more partial solutions and hence a combinatorial number of contexts. Exploration of all the possible paths in the solution space.
and maintenance of all the contexts are required if an ATMS is to be used. Because CDSP problems typically have combinatorial solution spaces, it is very inefficient to develop and maintain all possible solutions. Also, during expansion of each state, the problem solver has to check the extensions of $CS$ from the existing states against local constraints and their schedulability. This gives added complexity to the search. Further, some paths explored by the problem solver may not be eventually preferred by the user. Hence, from the computational standpoint, the generation and maintenance of multiple partial solutions is computationally ill-suited for CDSP problem solving.

Further, there are representation problems for multiple possibilities in the ATMS structure. Labels cannot be captured in single environments because of the disjunctive nature of consequences which result from application of each inclusion constraint (see Figure 7.3). Consequently, labels will not be able to be represented as a single set of environments and the label propagation mechanism of ATMS will not work for CDSP problem solving because application of same constraints to different contexts may result in addition of multiple conjunctive support structures. Also, revision in any assumption would require propagation of labels through testing of $CS$ in the labels against local constraints in subsequent states. If a multiple-context ATMS were used, it would require some sort of a multiple state node representation of knowledge in the ATMS network. This would allow the conjunctive support structures (as assumptions) selected to satisfy each inclusion constraint in each of the multiple problem solving paths to be explicitly represented. These assumptions would form justifications for inclusion constraints and might be different for the same constraints in different search paths. The multiple state node representation of ATMS knowledge adds to the computational complexity in the problem solving. Hence, the multiple-context ATMS use in the problem solving seems to be much more of a hindrance than a help although the other features of the ATMS are needed. The inability to capture and reason with all possible solutions necessitates use of additional control. Some form of backtracking is needed in the problem solving. (CDSP problems seem to be easily solvable in a dynamic environment using the multiple contexts scenario if the computations can be done in parallel. Parallel constraint satisfaction for CDSP problem solving is beyond the scope of this thesis.) We need to adopt a strategy which focuses problem solving allowing only promising partial solutions to be considered and maintained.

In light of the above drawbacks in using a multiple-context ATMS in the incremental and user-interactive solving of CDSP problems, we present, in the following sections, an approach which exploits a single-context ATMS in conjunction with the dependency-
7.5. **IPLAN and ATMS**

directed backtracking as the problem solver control.

### 7.5.3 Structure and Role of CDSP-ATMS

In the dynamic CDSP problem solving, we need a single-context ATMS to record the justification structures to facilitate dynamic reasoning with constraints and provide semantics for traditional label propagation, inconsistency detection and resolution. Some mechanism is needed to record the single context information so that dynamic reasoning with constraints is facilitated. This is achieved by adapting the multiple-context ATMS to a single-context ATMS (we call it CDSP-ATMS). The IPLAN problem solver communicates the problem solving data to the CDSP-ATMS and accesses information from it as required through an interface. Dependency-directed backtracking is invoked if a dead-end situation arises. The CDSP-ATMS is purely used for house-keeping during problem solving. It caches the justification information including nogoods and facilitates communication of recorded information with the problem solver. CDSP-ATMS is void of any deductive power. The problem solver performs and controls all the house-keeping operations, including label propagation.

Here, we present the structure of our CDSP-ATMS and we demonstrate how the IPLAN problem solver communicates with it in CDSP problem solving. The use of a single-context ATMS requires some mechanism to record a single context in the multiple context ATMS. This recording is achieved by using the notion of states in our CDSP-ATMS structure and using special nodes (called *state nodes*) to capture problem solving state knowledge. A state node records a label for the variables assigned so far, the justification for the state node, and if the label is not empty, a schedule is found for the aggregated structure $CS$ derived from the assignments in the label. If a schedule cannot be found for a non-empty label then schedule is assigned $fail$. On the other hand, if the label is empty then schedule is assigned $nil$. A CDSP-ATMS state node is thus

$$<\text{state-node}, \text{label}, \text{justification}, \text{schedule}>$$

where *state-node* is a unique state identifier, and *label* and *justification* have the conventional ATMS semantics. In CDSP-ATMS a label has a single environment and is represented as a set of assumptions such as \{ic1:css, ic2:css, ic3:css, ...\}, where $ic1$, $ic2$ and $ic3$ and so on denote the inclusion constraints applied and the corresponding csss are the conjunctive support structures found to satisfy the respective constraints with respect to the $CS$ structure constituted by the label in the previous state. (Note,
CSS is a set of tuples defining a conjunctive support structure. In this two-tier representation, if an inclusion constraint (say ci) is already satisfied by the aggregation of csss in the existing state then in the new state, it is represented as ci:{}. Justification records the justification for the state node. It is given by the tuple (state-node cj:css). The order in which inclusion constraints are applied is captured in the justification structure and is easily derivable from it.

The other nodes in the CDSP-ATMS are the premise, assumption and contradiction nodes. They use the conventional ATMS representation except that labels are singleton environments.

All constraints are treated as premise nodes and represented as <constraint, {}, {}>, where constraint is the constraint datum and there are no labels and no justifications recorded for constraints. The differentiation between constraints and their use in search are determined by the problem solver. For example, the problem solver knows that preferential constraints are defeasible and allows their retraction.

Every assignment of a conjunctive support structure to a constraint variable (also called a decision) is treated as an assumption node and represented as <c:css, {c:css}, {c:css}>. Each of these assumptions along with the current state node form justifications for the next state node. Note, as in ATMS, justification in CDSP-ATMS structure also implies logical consequence.

All inconsistent environments are nogoods. These environments are made to justify the distinguished contradiction node ⊥ which represents falsity. The contradiction node has an empty label. It is represented as <⊥, {}, {}>.

Application of an inclusion constraint to some existing state creates a new state which is justified by the applied constraint and the old state. If the problem solver discovers a CS constituted in a set of variable assignments to be inconsistent then the set of plan objects contained in CS is recorded with all the violated constraints in the nogood database. The assumed variable assignments and the violated constraints are made to justify the contradiction node ⊥. For example, if CSi violates the constraints C1, C2 and C3 then the nogood database contains ({{C1, C2, C3}, CSXi}), meaning that C1 ∧ CSXi → ⊥ or C2 ∧ CSXi → ⊥ or C3 ∧ CSXi → ⊥. CSXi is the set of plan objects contained in the csss of the variable assignments. The recording of violated constraints with the assignments are necessary because in the event of retraction of a constraint, the nogoods database has to be updated with respect to the retracted constraint. If the retracted constraint is the only constraint in a nogood tuple then the tuple is removed from the database. Otherwise, if there is a non-singleton set of
constraints containing the retracted constraint then the retracted constraint is removed from the tuple.

Each time a variable assignment is found, all the constraints are checked and no-goods recorded. If the values assigned to a set of constraint variables (i.e., the cumulative structure resulting from the aggregation of values assigned to the constraint variables) is consistent then the label of the corresponding state node must contain the label with an environment consisting of the assignment assumptions. This is achieved by creating a new state node for each applied inclusion constraint and appropriately justified. During problem solving, if $CS$ is found to be inconsistent then the set of assignment assumptions responsible for the inconsistency are declared to be inconsistent. All the constraints they violate are recorded with them as nogoods. They are made to justify $\bot$, the contradiction node. They are checked during search for each variable assignment and avoided. This avoids re-discovery of already known inconsistencies. Like the ATMS, CDSP-ATMS can only represent consistent contexts. An empty label at a state node shows that the context is inconsistent with respect to non-scheduling constraints. In addition, if the $CS$ captured in a label is non-schedulable then the label is left intact but the schedule in the state is marked $\text{fail}$.

As an example of the CDSP-ATMS structure, consider Figure 7.4. If a plan already exists then the initial CDSP-ATMS state node $Sn0$ is created and justified by $\text{plan:}CS$, where $CS$ is the cumulative structure of the plan. Otherwise, the problem solver chooses the first inclusion constraint $ic0$ for application. A conjunctive support structure $css$ is found that satisfies $ic0$. The pair $ic0:css$ is an assumption and forms the justification for the initial state node $Sn0$. The label propagates to $Sn0$ if all the constraints are not violated. The problem solver then selects the next best possible inclusion constraint $ic1$ for application based on information contained in $Sn0$. $ic1$ is applied, in a similar manner to an inclusion constraint application in Model-Search, by finding through heuristic-based guidance and checking against nogoods a $css$ which, when added to the previous assignment as captured in the $Sn0$ label, satisfies $ic1$. In some cases, an inclusion constraint might already be satisfied in the existing context such as $ic3$ is already satisfied in $Sn2$ in the above figure. No $css$ assignment is hence made to the respective constraint variable.

The informed problem solving continues with progressive extensions of the CDSP-ATMS network. As an inconsistent context results in problem solving, steps are taken to exploit the dependency information recorded in the network to recover from it. For example, in the above figure, application of $ic4$ to $Sn3$ results in the violation of $C$
Chapter 7. Incremental Planning: IPLAN

Symbols

- Justification
- $n$ CDSP-ATMS state node
- $x:y$ Assumption - constraint:assignment pairs
- $\text{constraint}$ Premise - a constraint node

Figure 7.4: An Illustration of a CDSP-ATMS Structure.
where $C$ can be any active non-scheduling constraint. When this inconsistency is detected, the problem solver creates a new node, $S_{n4}$ in this case, and assigns it an empty label. It also finds which assignments have been responsible for the inconsistency. In the example, they are $ic4:css$ and $ic2:css$. The problem solver uses DDB to revise these assignments to recover from the inconsistency. Recalling, a schedule is found by the standard backtrack search. If a $CS$ is not schedulable then the existing scheduling engine does not detect which scheduling constraints have been violated for each configuration of plan objects. Thus, the culprit assumptions causing the violations cannot be revised. The current problem solver records the plan objects in $CS$ with the violated set of scheduling constraints as a nogood.

In general, on the application of an inclusion constraint, the problem solver checks the satisfaction of all the constraints and records any inconsistencies. The scheduling constraints are checked by finding a schedule for the aggregation of the cumulative structure $CS$ constituted in the current state node and the chosen $css$. If there are any inconsistencies, then the violated constraints and the assignments are made to justify $\bot$, the resulting $CS$ is recorded with the violated constraints as nogood, and $\emptyset$ is recorded as the label of the next state node. Determination of which assignments lead to violations is easily done by first finding the plan objects that result in the violations and then checking the source of these culprits from the latest environment—the environment which resulted in the empty label. This information allows blaming of previous decisions which lead to failure and facilitates recovery from failure through a dependency-directed revision.

The CDSP-ATMS upholds most of the properties of the de Kleer type ATMS. The adaptations include a new representation for nogoods, and the introduction of the special state nodes which forces representation of single contexts. Since, the CDSP-ATMS does not capture all solution possibilities, it does not facilitate revision and exploration of solutions by simple context switching intrinsic to de Kleer’s ATMS. Revision and the exploration are facilitated by CDSP-ATMS through DDB. Retraction and addition of assumptions are required in the CDSP-ATMS whereas they are irrelevant to the ATMS. Also, all the house-keeping operations in the CDSP-ATMS are performed by the problem solver and CDSP-ATMS does not have any deductive power.

The CDSP-ATMS structure captures the solution structure $SSeq$ in the IPLAN problem solving. $SSeq$ for a sequence of constraints $<C_1, C_2, \ldots, C_n>$ is easily derived from the CDSP-ATMS state-node sequence $<S_0, S_1, \ldots, S_n>$. 

7.5. IPLAN and ATMS

165
Chapter 7. Incremental Planning: IPLAN

7.6 A Summary of IPLAN Strategy

Like in Model-Search, in IPLAN, the problem solver focuses on a single context, the current set of variable assignments. The assignments are represented in the CDSP-ATMS states as labels succinctly capturing the incrementally developed CS up to the current state. The search is driven through application of inclusion constraints one context at a time.

In the IPLAN approach, the problem solver accepts a batch of preferential constraints (usually one) at a time from the user. The constraints are functionally differentiated and appropriately used in a similar way to their use in Model-Search. The difference is that in Model-Search all the constraints are known prior to problem solving and only one solution is found whereas in IPLAN, only the formal constraints are known prior to problem solving, and the preferential constraints are dynamically added and applied and the solution progressively developed. If an application results in inconsistency then either the user directs recovery from it or recovery is done through the DDB mechanism which uses the knowledge recorded in the CDSP-ATMS network. The user is permitted browsing of partial solution states, seeking of advice (if required), and directing resolution of inconsistencies which arise from user-added constraints. The search is driven and directed through user formulation of a new constraint from each existing state. Hence, in IPLAN, the user is given more control and flexibility than in Model-Search. The DDB control of IPLAN and the coupling CDSP-ATMS representation of problem solving knowledge supports this pragmatic and exploratory style of problem solving.

The objective of IPLAN is to progressively and systematically search for a solution with cooperation from the user. During the interactive problem solving, if an empty label results for a state node in CDSP-ATMS, then DDB attempts to recover from the failure. On the other hand, if the label is non-empty but the schedule is assigned fail then the previous state node and the added constraints are displayed to the user and a relaxation of the constraint elicited from the user. This user guidance is required because there is currently no automatic mechanism in IPLAN that allows detection and revision of culprits which cause non-schedulability of a given CS.

IPLAN has been developed with the view that the user will interactively and systematically guide search for a solution. To give an added flexibility in the user-directed search, IPLAN supports an algorithm for automatically extending an IPLAN partial solution to a solution. This is to address the situation that at any point in the user-directed search, the user preferences may get exhausted without a solution being arrived
7.7. Reasoning with Constraints in IPLAN

at. From this point the user may direct an automatic search for a solution. The automatic search invokes the model-search1 search engine on the partial solution and finds a Model-Search solution if there is one. If a solution is found then the solution path resulting from model-search1 is transformed to CDSP-ATMS representation and the existing CDSP-ATMS structure extended with it. (The extended CDSP-ATMS structure is used for explanations and answering queries.) However, if a solution is not found, then CS is not tenable. The untenability may be because each extension to CS results in violation of at least one formal non-scheduling constraint or it is non-schedulable. While assumptions causing the former can be identified and revised, because all the scheduling constraints are active on the whole schedule, it is difficult to identify which particular assumptions cause the inconsistency and need to be revised. Therefore, DDB cannot be used in the revision to recover from schedule constraint inconsistencies.

Finally, if a solution is not found from the above then model-search1 is executed anew for an empty CS, an empty schedule, and $\Delta = \text{Form-Cons} \cup \text{Pref-Cons}$. If the solution is still not found then $\Delta$ is declared unsatisfiable.

In summary, the following control regimes are employed in the IPLAN approach until a solution is found or all regimes used.

1. User preferences are formulated taking advice from the system, and are processed using DDB.

2. Application of model-search1 on existing transition point in problem solving.

3. Application of model-search1 on $\Delta$ and guided by an existing nogoods database.

   If a solution is found then it can be revised through user preferences and DDB.

These regimes allow progressive development of a solution or a heuristic-guided search for one solution for a given set of constraints followed by user-directed revisions if any. In degree planning, usually regime 1 finds a solution. Only when a consistent schedule cannot be achieved from regime 1, regime 2 is used.

7.7 Reasoning with Constraints in IPLAN

The user drives IPLAN by interactively communicating preferential constraints and other instructions to it. The preferential constraints are appropriately processed based on the functional categories to which they belong. The inclusion constraints effect the elucidation of solutions. Their application is guided by information contained in the existing problem solving state, by the COBWEB knowledge clusters, and in
consultation with an existing nogoods database. It involves search for an assumption (a css) which together with the CS from the existing state satisfies the constraint and does not contradict any of the incompatibility constraints. All the non-scheduling constraints are checked on each constraint application, and the nogood database updated and DDB invoked for revision. Any unapplied formal inclusion constraints are finally used in the search through model-search1.

The preferential constraints are dynamically added in IPLAN problem solving. Their acquisition is partially directed by IPLAN. IPLAN supports addition of any kind of preferential constraint at any time during problem solving. But, because it is computationally less expensive to process the category preference constraints, and general exclusion constraints early in the problem solving, it is preferable to get them as early as possible.

We outline below the behavior of IPLAN in reasoning with each category of preferential constraints (see Table 5.1 for a general categorisation of constraints). We assume a CDSP-ATMS structure is given and $Sn_i$ represents the latest state node in the structure. Also, we refer to the cumulative structure captured in the label of $Sn_i$ as $S_{ni}.CS$ and a conjunctive support structure as css. The computational descriptions given below are captured in an algorithm for IPLAN presented later.

**7.7.1 Inclusion Constraints**

The preferential inclusion constraints supported in the current version of IPLAN are INCLUDE $<\text{plan-object}>$, and the category preference constraint TAKE-OBJ-FROM $<\text{class}>$. Each of these constraints are added at each state in IPLAN problem solving. TAKE-OBJ-FROM $<\text{class}>$ is acquired preferably at the start of the problem solving.

**INCLUDE $<\text{plan-object}>$**

This inclusion constraint $ic$ is tested against $S_{ni}.CS$ and if it is already satisfied then $\{\}$ is recorded as the assumption css for the satisfaction of $ic$. Otherwise, a css is found which, when added to $S_{ni}.CS$ satisfies $ic$. This search avoids any css which, when added to $S_{ni}.CS$ is subsumed by a nogood. Also, the unique support property of plan objects and the incompatibility constraints have to be satisfied by the extended CS. The heuristics used in Model-Search in selecting the css are also used here. The found css and the $ic$ are then communicated to the CDSP-ATMS structure. The pair $ic:css$ (denoting $ic$ is satisfied by the aggregation of css and the existing cumulative structure $S_{ni}.CS$) is recorded in the CDSP-ATMS structure as an assumption. The
next state node $S_{n+1}$ is created and justified with this assumption and the previous state node $S_n$. $S_{n+1}.CS$ is then checked against all non-scheduling constraints. If any of these constraints are violated\(^3\) then the culprit plan objects and the csss to which they belong are identified from the label and $S_{n+1}$’s label is set to empty. The violated constraints and the set of plan objects in $S_{n+1}$ are recorded as nogoods. The culprit assumptions are revised, either automatically through DDB or under guidance from the user, until the resulting state node has a non-empty label.

In the creation of $S_{n+1}$, if $S_{n+1}$ has a non-empty label (i.e., if $S_{n+1}$ does not violate any non-scheduling constraints), then the schedulability of $S_{n+1}.CS$ is checked. This is done by invoking the scheduler on $S_{n+1}.CS$. A schedule is found if one exists. If a schedule cannot be found then $S_{n+1}.schedule$ is marked fail, the user is asked to relax the previous constraint and the problem solver backtracks to the previous state and resumes problem solving from there.

In each case of failure, the user is notified, and new constraints and instructions are sought from the user. This run-time input may be another constraint, request for advice and input of another constraint, retraction of an existing constraint, a signal to automatically recover from failure and continue search, or exit from search. In the automatic continuation, the problem solver backtracks to the most recent state which has the justification causing the failure, and revises the culprit assumption by finding another css. If multiple inclusion constraints are violated then the dependency-directed control backtracks to the most recent state containing one of the constraints and re-satisfies it by revising its assignment.

DDB allows jumping to new states and continuation of search from the new state. In this scheme, the nogoods are incrementally built up and the previous path traversed from the backtracked point are no longer useful and hence purged. All the “undone” preferential constraints become unapplied in the backtracking. These constraints are automatically applied from the new state creating CDSP-ATMS state nodes and extending the CDSP-ATMS structure in the process. The interactive search continues until a non-empty label is found for the application of all the preferential inclusion constraints.

Retraction of the non-category inclusion constraints results in the following behaviour. The retracted constraint is added to a set of retracted constraints OUT-CONS globally maintained by the problem solver. The assumption to satisfy the constraint is

\(^3\)Note the only non-scheduling constraints which can be violated are the inclusion constraints as the others: the exclusion and incompatibility constraints, are used in guiding selections of conjunctive support structures for satisfying the inclusion constraints.
Chapter 7. Incremental Planning: IPLAN

retained in the CDSP-ATMS structure until there is a need to backtrack to the state node justified by the assumption or an earlier state node. In these cases, the retracted constraint is not applied. Retraction of a non-category preference inclusion constraint does not have any other effect on the problem solving. For example, from Figure 7.4, if at state \textit{Sn2 ic1} is retracted by the user then \textit{ic1} is recorded as an OUT-CONS and the problem solving continues. In future if there is a need to backtrack to any state node at or below \textit{Sn1} (i.e., \textit{Sn0} or \textit{Sn1}) then all the constraints in OUT-CONS, including \textit{ic1}, are not applied.

\textbf{TAKE-OBJ-FROM <class>}

This constraint specifies that rather than selecting plan objects randomly to satisfy any aggregate constraints, the plan objects from certain specified class are to be selected. These plan objects get selected through the \textit{get-bias-seq} function. Since the current version of IPLAN does not support aggregate preferential constraints, this constraint gets used only by \textit{model-search1} as in Model-Search. The constraint is acquired at the start of problem solving and any changes in the category preference is achieved at run-time using CHANGE-CATEG-PREF constraint (as discussed later in §7.7.5) from Change-Cons.

Retraction of this constraint removes the restriction on selection of plan objects as specified in the constraint for future selections.

7.7.2 Exclusion Constraints

There are two kinds of exclusion constraints support in IPLAN: EXCLUDE \textit{ <plan-objects >} and the category preference exclusion constraint NOT-TAKE-OBJ-FROM \textit{ <class>}. The latter is preferably added prior to the start of problem solving and the former is added at any state during problem solving.

\textbf{EXCLUDE <plan-objects>}

If the assumptions captured in the label of some existing state are inconsistent with these constraints then these assumptions are recorded with the contradicting constraints as nogoods. Then, DDB is invoked to the \textit{least} recent state node which has its justifying assumption causing the inconsistency. An alternative assumption consistent with the constraint is chosen and the problem solving continues from the backtracked point. Further, the exclusion constraint is treated as a local constraint and checked each time an inclusion constraint is applied.
NOT-TAKE-OBJ-FROM <class>

This constraint specifies that plan objects from certain specified attribute values are not to be included in the solution being searched. If $S_n_i$ already contains plan objects with these attribute values then the assumptions containing them are identified and revised using DDB. The working set of plan objects $WS$ is filtered with the plan objects pertaining to the class so that in future selections of $css$, those $css$ are selected which do not contain any of these plan objects.

Retraction of the exclusion constraints does not affect the existing solution but in future applications of inclusion constraints, all the plan objects specified for exclusion in these constraints are made available. This is achieved by updating the existing working set of plan objects $WS$ updated with the plan objects made available by the retraction.

7.7.3 Temporal Constraints

The preferential temporal constraints are:

- PLAN-TIME <pos-integer><temporal-class>,
- PLAN-TIME-BETWEEN <pos-integer><temporal-class> AND <pos-integer><temporal-class>,
- TAKE-IN <plan-objects><temporal-interval>,
- TAKE-BEFORE <plan-object1><plan-object2>, and
- TAKE-CONC <plan-object1><plan-object2>.

All these constraints are globally active on the scheduling process which is performed in the generation of each successor state node—in fact all the temporal constraints are processed at each state locally during the scheduling process. In the creation of a new state node, the schedulability of the structures in the assumptions are checked by finding a schedule for their combination. If a schedule cannot be found then schedule is marked fail in the new state. The search for a schedule in the current model uses the standard backtracking strategy and hence, does not allow exploration of multiple possibilities in the search and blaming of decisions to help in revision. Instead, an exhaustive search is performed for a schedule and if a schedule is not found then the user is asked to retract the latest constraint.
Given a state with a non-empty label, addition of any of the above temporal constraints will invoke the scheduler to search for a configuration for the plan objects contained in the structures of the label, which satisfies all the scheduling constraints \textit{Sched-Cons} and respects the support constraints captured in those structures. If a schedule cannot be found, then the scheduling constraints and the \textit{CS} in the current label are together recorded as a nogood, \textit{schedule} in the current state node is marked \textit{fail}, and the user is advised that the added constraint leads to inconsistency and is asked to relax the latest temporal constraint. However, if the temporal constraint results in a schedule then it is accepted, the set of temporal constraints is updated with the new constraint and the updated set is used in future schedulings.

Also, at some state node, given a set of temporal and load constraints, a given \textit{CS} may not be schedulable. Because the causes of inconsistency cannot be inferred from the scheduling process in the current model, dependency-directed revision cannot be performed. The user is asked to relax constraints from the existing set of preferential constraints and the problem solving continues. If the user seeks a brute-force solution then \textit{model-search1} is invoked.

Retraction of a temporal constraint does not affect a solution. The retracted temporal constraints are deleted from the set of \textit{Sched-Cons} and not used in any future scheduling.

In summary, reasoning with the changing temporal constraints is partially addressed in \textit{IPLAN} particularly, in light of the kinds of some simple temporal constraints which pertain to the degree planning problem. In the cases where schedules cannot be found, \textit{IPLAN} does not identify which constraints cause the failure. This is useful information in guiding recovery from failure. An amelioration of the current approach to scheduling in regards to an informed recovery form failures in search for schedules is proposed in the suggestions for further work in Chapter 9.

### 7.7.4 Conditional Constraints

\textit{IPLAN} allows what-if analysis through a run-time addition of conditional constraints. These constraints are conditional queries in regards to an existing problem solving state (see Table 5.1 for a summary). The problem solver behaves differently in processing each different kind of query made. In all the cases, first, a copy $SnC_i$ of the latest state node $Sn_i$ is made. Also, after reasoning with the conditional constraint, if the conditional changes are accepted by the user then $Sn_{i+1}$ is created capturing the results from the application. Otherwise, the control returns to $Sn_i$ and further problem solving
If the constraint is WHAT-IF INCLUDE \(<\text{plan-object}\>\) or WHAT-IF EXCLUDE \(<\text{plan-objects}\>\) a new state node is created from \(SnC_i\) and the contents displayed to the user. WHAT-IF SWAP \(<\text{plan-object1}> <\text{plan-object2}\>\) results in the application of EXCLUDE \(<\text{plan-object1}\>\) and INCLUDE \(<\text{plan-object2}\>\) in this order creating extensions to the CDSP-ATMS structure containing \(SnC_i\). On the other hand, if it is a scheduling constraint, a schedule is searched for the \(CS\) derived from the label of \(SnC_i\) and displayed to the user.

After processing each of the conditional constraints, the user is allowed to accept the conditional constraints as respective non-conditional constraints. If the constraints get accepted, the CDSP-ATMS structure gets extended by the \(SnC_i\) node with the appropriate justifications to \(S_{n+1}\).

### 7.7.5 Change-Cons Constraints

The Change-Cons constraints are constraints on problem solving control. SWAP \(<\text{plan-object1}> <\text{plan-object2}\>\), KEEP-ALL-BUT \(<\text{plan-objects}\>\) and CHANGE-CATEG-PREF are the constraints supported in IPLAN. These constraints are added by the user, as desired, at the problem solving states.

**SWAP \(<\text{plan-object1}> <\text{plan-object2}\>\)**

This constraint splits to EXCLUDE \(<\text{plan-object1}\>\) and INCLUDE \(<\text{plan-object2}\>\).

**KEEP-ALL-BUT \(<\text{plan-objects}\>\)**

This constraint specifies exclusions of certain plan objects, given by \(plan-objects\), with respect to an existing state node \(Sn_i\). It is satisfied by marking all the plan objects in \(plan-objects\) as objects for exclusion and all the other plan objects in the existing label as objects for inclusion. The problem solver then backtracks to the least recent state node which is justified by a structure containing an object from the the set \(plan-objects\) and resumes search with the new batch of constraints. The search continues automatically until all the existing preferential constraints and all of the new inclusion constraints are satisfied. All the exclusion constraints are used locally to guide selection of the csss. This constraint allows minimal revisions of existing solutions.
CHANGE-CATEG-PREF

The category-preference constraints are NOT-TAKE-OBJ-FROM <class> and TAKE-OBJ-FROM <class>. They are usually at the start of problem solving. They are changed by CHANGE-CATEG-PREF during problem solving. Revision of category preferences is performed by retracting old category preferences and adding new ones. Retraction does not change an existing solution. Addition of a new category constraint, however, requires backtracking to Sn0 and resumption of an automatic search from there until all the preferential constraints are satisfied.

7.8 Explanation and Advice Generation

Given some CDSP-ATMS structure at some problem solving state, WHY <plan-object> extracts from the CDSP-ATMS structure justifications for the presence of plan-object in the state and explains it. Also, at some state, the user may seek advice from the system. Simple advice is generated by IPLAN. From the Case-Info all instances of plans which satisfy the existing set of preferential constraints are retrieved and the differences from the existing partial solution suggested to the user. Any of these differences when added to the existing partial solution would carry it to completion.

Also, IPLAN finds for the plan objects in the existing state, their consequent set Conseq-Set, and relevant COBWEB clusters, from which it determines the plan objects to include in the solution so that the unapplied inclusion constraints are satisfied. Rather than arbitrarily choosing the csss, this advice allows the user to select csss containing objects of his/her preference.

7.9 IPLAN Algorithm

Here, we present the IPLAN algorithm. It combines a dependency-directed backtracking control and the single-context CDSP-ATMS to capture the interactive, incremental and exploratory nature of problem solving as required for the IPLAN formulation of CDSP problems.
7.9. IPLAN Algorithm

IPLAN

iplan: () → Schedules ∪ XPLANS ∪ {fail}

Variables

Plan-Objs: A set of all plan objects.
WS: A working set of plan objects.
Form-Cons: A set of formal constraints.
Pref-Cons: A set of preferential constraints.
id: A unique plan identifier.
acen: A constraint.
new-cons: A constraint.
categ-pref: A set of category preference constraints.
viol: A set of violated constraints.
cs: A cumulative structure.
css: A conjunctive support structure.
schedule: A schedule—reverse of a plan.
xplan: An executed plan.
nogoods: A set of nogoods.
change-cons: A run-time constraint.
retract-cons: A run-time retraction constraint.
ac-cons: Accumulated constraints in a cs.
CUL: A set of culprit assumptions.
OUT-CONS: A set of constraints retracted at run-time.
viol: A set of violated constraints.
cdsp-atms: A CDSP-ATMS structure.
Sn0: Initial CDSP-ATMS state.
Sg: Model-Search Goal state.
assumption: A CDSP-ATMS assumption node.
cn, nn: CDSP-ATMS nodes.

cn.schedule: A schedule in the cn node.

cn.label: Label of the CDSP-ATMS state-node cn.

cn.justification: A set of assumptions which justify cn.

assumption.label: Label of an assumption CDSP-ATMS node.

LStruct: A COBWEB cluster.

Case-Info: A database of case objects.

nid: Unique node identifier.

SP: A Model-Search solution path—a sequence of state nodes.

{Initialisation}

Input id, Pref-Cons, Form-Cons, CS, xplan, schedule.

(*Pref-Cons, CS, xplan, schedule may already exist for id.*)

user-satisfied ← false  (*Flag*)

OUT-CONS ← {}  (*Run-time retracted constraints*)

nogoods ← (get-plan-obj-sets (get-typel-incomp-cons Form-Cons))  (*Nogoods from Type 1 incompatibility constraints.*)

{Set-Up Data}

if ∃ schedule for id then

if (evaluate-plan xplan) = pass then (return xplan) fi

if (not (verify-plan schedule xplan)) then schedule ← nil

(*schedule is inconsistent with Form-Cons.*)

(*schedule is consistent but further revision may be required.*)

fi

else

schedule ← nil

Pref-Cons ← nil

CS ← nil

xplan ← nil

fi

{Set-Up CDSP-ATMS Structure and Initial Node}
7.9. IPLAN Algorithm

\texttt{cdsp-atms} ← (create-cdsp-atms)

\textbf{if} \textit{cs} \neq \textit{nil} \textbf{then}

(*Set up the initial state node with an existing plan.*)

\textit{assumption} ← (create-assumption-node \textit{schedule}:\textit{cs})

(*\textit{schedule} exists prior to problem solving, \textit{cs} is the existing cumulative structure.*)

\textit{cn} ← (create-state-node \textit{Sn0})

(*Create initial CDSP-ATMS node.*)

\textbf{if} (satisfies? (get-sched-cons (\textit{Form-Cons} \cup \textit{Pref-Cons}) \textit{schedule}) \textbf{then}

\textit{cn.schedule} ← \textit{schedule}

\textbf{else} \textit{cn.schedule} ← (schedule \textit{cs} (get-sched-cons (\textit{Form-Cons} \cup \textit{Pref-Cons})

(get-unused-intervals \textit{xplan}))

fi

(justify-state-node \textit{cn} \textit{assumption})

\textbf{if} (goal-state-node? \textit{Sn0}) \textbf{then} \textit{user-satisfied} ← \textit{true} \textbf{fi}

\textbf{else}

(*Set up initial state node in the CDSP-ATMS by acquiring and processing first preferential constraint.*)

\textit{cn} ← (create-state-node \textit{Sn0})

(*Create initial CDSP-ATMS node.*)

\textbf{A:} \textit{acn} ← (ask-user-for-constraints \textit{schedule} \textit{Pref-Cons})

\textit{Pref-Cons} ← \textit{Pref-Cons} \cup \{\textit{acn}\}.

\textbf{if} \textit{acn} is categ-preference \textbf{then}

\textit{categ-pref} ← \textit{categ-pref} \cup \{\textit{acn}\}

go to \textbf{A}

\textbf{elseif} \textit{acn} is exclusion \textbf{then}

\textit{WS} ← (apply-exclusion-constraints \{\textit{acn}\} \textit{Plan-Objs})

(*Filter from WS plan objects specified in \textit{acn}.*)

go to \textbf{A}

\textbf{else} \textit{nn} ← (apply-incl-cons \textit{acn} \textit{cn})

(*Heuristics are used to find from WS, a \textit{css}, addition of which to \textit{cn.cs} results in the satisfaction of \textit{acn}. The selection avoids nogoods, satisfies incompatibility constraints and satisfies the unique support property.*)

\textit{user-satisfied} ← \textit{false}

\textbf{fi}

\textbf{fi}

{\textbf{Main Loop}}

\textbf{until} \textit{user-satisfied} \textbf{do}
Chapter 7. Incremental Planning: IPLAN

if \( nn.label \neq {} \) and \( nn.schedule = fail \) then
(*Assumptions causing the failure cannot be found—user is asked to help.*)
until \( nn.schedule \neq fail \) do
  (update-browser (explain-failure \( nn \)))
  (explain-state-node \( cn \))
  \( acn \leftarrow (\text{ask-user-for-constraints } cn.schedule \text{ Pref-Cons}) \)
  \( nn \leftarrow (\text{process-cons } acn \space cn) \)
  (*Constraint is processed and CDSP-ATMS structure is updated.*)
endo
elseif \( nn.label = {} \) then
(*A CS that is consistent with non-scheduling constraints cannot be found.
Perform revision through dependency-directed backtracking: find the culprits and ask user to revise assignments if inconsistency arises from user preferences; or automatically revise if it arises from default assumptions.*)
  Identify the assumptions causing the inconsistency by propagating through the dependency structure.
  Identify the constraints violated by the assumptions.
  Update the nogoods database.
  if user-guided revision then
    display inconsistencies and ask user to relax constraints
  else (*The following steps describe automatic revision from dependency-directed backtracking.*)
    Identify state nodes which contain the culprit assumptions and record as a sequence \( CUL \) from most recent to least recent.
    \( B: \space cn \leftarrow (\text{pop } CUL) \)
    \( cnss \leftarrow \text{find the constraint which is the justification for a culprit assumption} \)
    \( nn \leftarrow (\text{apply-incl-cons } cnss \space cn) \)
    (*Nogoods are avoided in the application.*)
    if \( nn.label = {} \) or \( nn.schedule = fail \) then go to \( B \) fi
  fi
(explain-state-node \( cn \))
(*Display information from the current state node.*)
Ask user to select one instruction from Automatic-Search, Add-Constraint, Retract-Constraint, Change-Cons, Advise, Explain, Query, Exit.
if instruction is Retract-Constraint then
  \( retract-cons \leftarrow (\text{ask-user-for-retraction-constraints}) \)
  \( Pref-Cons \leftarrow Pref-Cons \setminus retract-cons \)
  \( OUT-CONS \leftarrow OUT-CONS \cup retract-cons \)
  (update-nogoods retract-cons)
7.9. IPLAN Algorithm

elseif instruction is Automatic-Search then

\[ S_g \leftarrow \text{(model-search1 cn.schedule (get-cs cn))} \]
('get-cs finds the cumulative structure contained in the label of current
CDSP-ATMS node. \( S_g \) is the search state returned by model-search1
through an exhaustive search of solution space.*)

if (goal-node? \( S_g \)) then (extend-cdsp-atms SP)
('Solution path in model-search1 is recorded in \( SP \) and the CDSP-
ATMS structure is extended with it if a solution is found.*)
else cn.label \leftarrow \{\}
('If a solution is not found then \( \Delta = \text{Form-Cons} \cup \text{Pref-Cons} \) is unsat-
sifiable, and the current state node label is marked empty.*)
fi

elseif instruction is Add-Constraint then

\[ \text{new-cons} \leftarrow \text{(acquire-constraint cn.schedule Pref-Cons)} \]
('acquire-constraint ensures that receipt of new constraints does not
result in Type-A inconsistency.*)

Pref-Cons \leftarrow \text{Pref-Cons} \cup \{\text{new-cons}\}

if (inclusion-cons? \text{new-cons}) then (apply-incl-cons new-cons cn)
('Apply the inclusion constraint and create a new state node.*)
elseif (exclusion-cons? \text{new-cons}) then (apply-excl-cons new-cons cn)
('Update WS and check if the exclusion constraint is violated by the
current state. If it is, then find the culprit from the least recent state
node through DDB, revise the assumption, and re-plan from that state
avoiding selection of any plan objects specified by the exclusion con-
straints.*)
elseif (temp-cons? \text{new-cons}) then (process-sched-cons cn)
('If \text{new-cons} is a temporal constraint then check whether the existing
schedule satisfies the new constraint. If it does not, then re-schedule the
\( CS \) from the existing state.*)
elseif (cond-cons? \text{new-cons}) then (process-cond-cons new-cons cn)
('Make a copy of the current state and reason with it in processing any
conditional constraints. The control returns to the existing state after
the reasoning.*)
fi

elseif instruction is Advise then

(update-browser (advise-user (get-plan-objs (get-cs cn)) Pref-Cons Form-
Cons LStruct Case-Info))

elseif instruction is Change-Cons then

\[ \text{new-cons} \leftarrow \text{(ask-user-for-change-cons)} \]
('Change-Cons constraints are ap-
propriately processed.*)

elseif instruction is Explain then display the justification structures in
the existing CDSP-ATMS structure and all the existing constraints
**elseif** instruction is Query **then** acquire user queries in the language supported in KMS and process them.

**elseif** instruction is Exit **then** quit.

if (goal-state-node? cn) **then** user-satisfied ← true

enddo

(explain cdsp-atms)
(*Explain the solution and answer any user queries in regards to the solution.*)

(save-plan-db id Pref-Cons cn.schedule (get-cs cn))

(return cn.schedule)

---

**apply-incl-cons: CONST × SN → SN**
Input: cns:CONST, cn:SN (SN is the domain of CDSP-ATMS state nodes)

{CDSP-ATMS State Expansion Engine}

if cn = nil then

   cn ← (create-state-node Sn0)
   css ← (get-css cns cn)
   (*Get an assumption that satisfies cns.*)
   assumption ← (create-assumption-node cns:css)
   assumption.label ← {cns:css}
   (justify-state-node cn assumption)
   cn.label ← {assumption}
   cn.schedule ← (schedule css (get-sched-cons (Form-Cons ∪ Pref-Cons) (get-unused-intervals z-plan)))
   (return cn)

else

   nn ← (create-state-node nid)
   (*A new state node is created with a unique node identifier nid.*)
   css ← (get-css cns cn)
   assumption ← (create-assumption-node cns:css)
   assumption.label ← {cns:css}
   (justify-state-node nn assumption)
   cs ← (get-cs cn)
   viol ← (get-violations (cs ⊕ css) (Form-Cons ∪ Pref-Cons ∪ (get-accum-cons css))
   (*viol is a set of all constraints violated by the accumulated assumptions.*)
if $\text{viol} = \{\}$ then

$\text{nn.label} \leftarrow \text{cn.label} \cup \text{assumption.label}$

$\text{ac-cons} \leftarrow \text{(get-sched-cons (Form-Cons} \cup \text{Pref-Cons)} \text{(get-unused-intervals xplan))}$

sched $\leftarrow \text{(extend-schedule css cn.schedule ac-cons)}$

if sched $= \text{fail}$ then

$\text{nn.schedule} \leftarrow \text{(schedule (css} \oplus \text{cs)} \text{(get-sched-cons (Form-Cons} \cup \text{Pref-Cons)} \text{(get-unused-intervals xplan))}$

fi

else $\text{nn.label} \leftarrow \{\}$

fi

(justify-state-node $\text{cn assumption}$)

if $\text{cn.label} = \{\}$ then

(justify-contradiction-node $\text{viol}$)

(update-nogoods $\text{viol}$ (get-plan-objs css))

fi

(return $\text{nn}$)

CDSP-ATMS Interface

In the above algorithm, the problem solver interfaces with the CDSP-ATMS structure through the functions summarised in Table 7.1.

<table>
<thead>
<tr>
<th>create-cdsp-atms</th>
<th>create-assumption-node</th>
</tr>
</thead>
<tbody>
<tr>
<td>create-state-node</td>
<td>justify-contradiction-node</td>
</tr>
<tr>
<td>justify-state-node</td>
<td>goal-state-node?</td>
</tr>
<tr>
<td>get-cs</td>
<td>explain</td>
</tr>
<tr>
<td>explain-node</td>
<td>set-node-datum</td>
</tr>
<tr>
<td>get-node-datum</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.1: Interface functions for CDSP-ATMS.

create-cdsp-atms returns a data structure which captures the entire state of a CDSP-ATMS. create-assumption-node takes an inclusion constraint conjunctive support structure pair and returns an assumption node with cns:css as the datum of the node. create-state-node creates a state node with a unique node identifier for the CDSP-ATMS structure. justify-contradiction-node justifies the contradiction node with all constraints and assignments that lead to inconsistency. justify-state-node justifies
a state node with assumption and state nodes. _explain-node_ takes a state node and explains its contents. _explain_ takes the existing CDSP-ATMS structure and explains the information contained in justification structures. _get-es_ takes a state node and retrieves the cumulative structure contained in it. _goal-state-node?_ takes a CDSP-ATMS state node and checks whether the cumulative structure contained in it satisfies all the constraints, including checking for its schedulability. If all the constraints are satisfied then the user is asked whether he/she is satisfied with the schedule contained in the state. _goal-state-node?_ returns _true_ if the user is satisfied. Otherwise, it returns _false_.

All node properties in CDSP-ATMS are set and retrieved by _set-node-datum_ and _get-node-datum_ respectively. The latter two functions are respectively represented by _<node>.<datum> f- <value>_ and _<node>.<value>_ in the above algorithm.

### 7.10 Analysis of IPLAN

Incremental and exploratory problem solving are hallmarks of the IPLAN problem solving model. The single-context ATMS house-keeping structure (CDSP-ATMS) in conjunction with the dependency-directed backtracking control facilitate the dynamic nature of problem solving. This combination is better suited to CDSP problem solving than a multiple-context ATMS with no backtracking. The state nodes in CDSP-ATMS forces focusing on a single context and captures the ordering in problem solving.

IPLAN also addresses the important concern of satisfying a maximal subset of preferential constraints. This results from the property of user-directed problem solving that at any state in IPLAN problem solving, all the preferential constraints defined upto the state are satisfied and the partial solution at the state is consistent with all the formal constraints. As soon as an added preferential constraint gives rise to an inconsistency, this property of preferential constraints and user-directed problem solving allows satisfaction of a maximal subset of preferential constraints simply by finding a solution from the previous state. The set of preferential constraints in the prefix of the preferential constraint sequence will be the maximal set of constraints that is satisfiable from the existing preferential constraint sequence. That is, if an IPLAN state _SN_i_ has a preferential constraint sequence _<C_1, C_2, ..., C_i>_ and if addition of a preferential constraint _C_{i+1}_ (i.e., _<C_1, C_2, ..., C_i, C_{i+1}>_) leads to an inconsistency then the set of constraints in the prefix of _C_{i+1}_ (i.e., _{C_1, C_2, ..., C_i}_) is the maximal set of preferential constraints that is satisfiable.

This property avoids the computationally expensive process of finding the maximal subset of preferential constraints that is satisfiable by enumerating the power domain.
of the set of preferential constraints and searching for one from the power domain.

The addition and retraction of constraints in IPLAN problem solving makes the IPLAN model nonmonotonic. This property allows user-exploration of solution space, revision of existing solutions and recovery from inconsistency.

The IPLAN problem solving is complete. It always finds a solution, if one exists, for a given set of formal constraints and a sequence of preferential constraints. The use of heuristics in choosing values for satisfaction of constraints makes development of partial solutions informed and search for a solution efficient. Also, an IPLAN solution is sound (i.e., it satisfies all the constraints).

The independence property of constraints make reasoning with constraints commutative. Addition of same preferential constraints in any order will result in a solution if there is a solution for the constraints. Also, application of constraints in search does not hinder the future applicability of other constraints.
Chapter 8

Degree Planning: A Case Study of CDSP Model

8.1 Introduction

This research on CDSP problem solving is problem-driven. The degree planning problem has been the motivation for it [142]. The CDSP model developed in the previous chapters caters for the problem-solving functionalities as required in the degree planning problem. In this chapter, we use the degree planning problem from the Australian National University (ANU) to illustrate the model. (The degree planning problem is explained and characterised in Chapter 1.) All knowledge pertaining to the domain is formalised and represented in the ORL framework, the degree evaluation and verification concerns are discussed, and results from experimentations of the CDSP model in this domain are demonstrated.

8.2 The Problem

Recalling, the objective of the degree planning problem is to find an acceptable plan\(^1\) of study for a student with various kinds of information given (see Figure 8.1). The acceptable plan has to be admissible (i.e., consistent with the formal constraints) and consistent with all the preferential constraints that are desired by the student. The formal constraints are stipulated in the legal requirements (as statutory law) for academic degrees, and all the CDSP properties discussed in Chapter 3 apply to the degree planning problem.

There have been a few attempts at developing automated systems for academic

---

\(^1\) Plans are also referred to as programmes in the degree planning terminology.
counselling [73, 164]. Both of these systems are rule-based. They do not treat the counselling problem as a constraint satisfaction problem, and they do not provide a language for the user to communicate his/her preferences and freely explore the solution space. Also, they do not support incremental problem solving, detection of constraint inconsistencies and their resolution, plan revision, hypothetical reasoning, plan verification and repair. Nachtsheim [109] reports on a formulation of a course scheduling problem as a CSP in the manner of *n-queens* problem. It performs standard backtracking search and does not provide for the human dimension which is essential in these problems from the points of view of acceptability of solutions and efficiency in search. None of the above approaches employ constraints as heuristics in guiding search. These approaches are *ad hoc*; they do not identify and use the problem properties (such as the plan object precedence and coincidence properties) in facilitating problem solving.

The academic counselling problem (which includes scheduling) belongs to a specialised but useful class of real-world problems which conform to the CDSP characteristics and can be solved by the CDSP model. Figure 8.2 describes its capturing by the CDSP problem solving model.
8.3 Knowledge Formalisation and Representation

Prior to illustrating the CDSP problem solvers in the degree planning domain, we present here the knowledge engineering aspect of the CDSP model. The formalisation and internal representation in ORL of knowledge from the ANU BSc degree domain (as given in [1, 2]) are illustrated from [141, 143].

8.3.1 Course Objects

A plan object in the degree planning domain is a *course* (or a *unit*). A course object is easily formalised and represented. An example of a course object is given in Table 8.1.

The input language CL allows declaration of sets of attribute names and their corresponding sets of attribute values. From the ANU BSc domain,

- Unit-Id is a set of unique unit identifiers and the identifier assigned to a course object has to be from the set Unit-Id,
- Groups = {A, B, C, D, E},
- Levels = {yr1, yr2, yr3, yr4, others},

![Figure 8.2: Degree Planning in CDSP Model.](image-url)
8.3. Knowledge Formalisation and Representation

Course Object:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit-code (Unit-Id)</td>
<td>Comp3031</td>
</tr>
<tr>
<td>Unit-name</td>
<td>Computer Science C31</td>
</tr>
<tr>
<td>Unit-title</td>
<td>Formal Languages and their Recognisers</td>
</tr>
<tr>
<td>Department (Departments)</td>
<td>Computer-Science</td>
</tr>
<tr>
<td>Group-number (Groups)</td>
<td>C</td>
</tr>
<tr>
<td>Level (Levels)</td>
<td>yr3</td>
</tr>
<tr>
<td>Weight (Wts)</td>
<td>.5</td>
</tr>
<tr>
<td>Prereq</td>
<td>(Comp2012 and (Sum-Wts (Mathematics Statistics) B \geq 1))</td>
</tr>
<tr>
<td>Co-Req:</td>
<td>{}</td>
</tr>
<tr>
<td>Incomp-Obj:</td>
<td>{}</td>
</tr>
<tr>
<td>Avail (Avail-Session)</td>
<td>1 session(s)</td>
</tr>
<tr>
<td>Duration</td>
<td>1 session(s)</td>
</tr>
</tbody>
</table>

Table 8.1: An Example of a Course Object in ORL.

- \text{Wts} = \{.5, 1, 2\} (also referred to as points in the ANU BSc degree domain),
- \text{Avail-Session} = \{1, 2, A\}, and

Also, the following sets and formal relations between them are declared in CL.

- \text{Temp-Int} = \{yr1s1, yr1s2, yr2s1, yr2s2, yr3s1, yr3s2, yr1, yr2, yr3\} where yr1s1, yr1s2, yr2s1, yr2s2, yr3s1 and yr3s2 are primitive temporal intervals, and yr1 = \langle yr1s1 yr1s2\rangle, yr2 = \langle yr2s1 yr2s2\rangle, yr3 = \langle yr3s1 yr3s2\rangle are composite temporal intervals.
- \text{Science-Departments} = \{Biochemistry, Botany, Chemistry, Computer-Science, Forestry, Geology, Human-Sciences, Mathematics, Physics, Psychology, Zoology\}.

- Economics-Commerce-Departments = {Commerce, Economic-History, Economics, Statistics}.

- Faculties = {Science, Arts, Economics, Law}.

### 8.3.2 Constraint Objects

The prerequisite, co-requisite and course availability constraints are captured in the course objects as supported by the CL syntax. The remaining constraints are represented as ORL objects using the schema of Table 4.4. Interpretation and formalisation of the legislative requirements in the ANU BSc domain are non-trivial tasks (see an extract of 1989 ANU BSc Order given in Appendix C). Problems encountered are largely due to syntactic complexity, ambiguity, vagueness, and correct interpretation and encoding of judgemental knowledge. In addition, some of the rules contain negative consequents. In these cases, an interpretation as negation as failure [22] is adequate. (Since, interpretation, formalisation and representation of legal knowledge for automated reasoning tasks are beyond the scope of the current research, these problems are identified and discussed with examples in [139]).

As an example, consider the rule OR21A from Appendix C. It is formalised and represented as the constraint object of Table 8.2. ?s and ?cons denote variables. ?s assumes either a Model-Search problem solving state or an IPLAN problem solving


Table 8.3: A Consistent Schedule.

<table>
<thead>
<tr>
<th>Final Schedule</th>
<th>Student Id: 9210042</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year 1 Session 1 : Comp1001 (wt: 4)</td>
<td></td>
</tr>
<tr>
<td>Year 1 Session 2 : Comp1002 (wt: 4)</td>
<td></td>
</tr>
<tr>
<td>Over Year 1 : Chem1001 Chem1002 Bot1001</td>
<td></td>
</tr>
<tr>
<td>Year 2 Session 1 : Bioc2001 Bot2004 Phil2080 (wt: 3)</td>
<td></td>
</tr>
<tr>
<td>Year 2 Session 2 : Bioc2004 Bot2002 Phil3053 (wt: 3)</td>
<td></td>
</tr>
<tr>
<td>Year 3 Session 1 : Bioc3005 Bioc3007 Bot3009 (wt: 3)</td>
<td></td>
</tr>
<tr>
<td>Year 3 Session 2 : Bioc3002 Bioc3008 Bot3005 (wt: 3)</td>
<td></td>
</tr>
</tbody>
</table>

8.3.3 Plans

The plans are represented as cumulative structures with explicit representation of supports and an accepted schedule drawn from the cumulative structure. For example, from Appendix D, the resulting cumulative structure is retained as \{(T Comp1002) (Comp1002 Comp1001) (T Phil3053) (Phil3053 Phil2080) (T Bot3009) (Bot3009 Bot2002) (Bot2002 Bot1001) (T Bot3005) (Bot3005 Bot2004) (Bot2004 Bot1001) (T Chem1002) (T Bioc3007) (T Bioc3008) (Bioc3007 Bioc2004) (Bioc2004 Bioc2001) (Bioc2001 Chem1001) (Bioc3008 Bioc2004) (T Bioc3002) (T Bioc3005) (Bioc3002 Bioc2001)\}. This structure allows revision in events of contingencies. The accepted schedule for this structure is given in Table 8.3.
XPLAN Object

<table>
<thead>
<tr>
<th>Isa:</th>
<th>XPLAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sid:</td>
<td>895043</td>
</tr>
<tr>
<td>Degree:</td>
<td>BSc</td>
</tr>
<tr>
<td>PEST:</td>
<td>(1 1989)</td>
</tr>
<tr>
<td>Results:</td>
<td></td>
</tr>
</tbody>
</table>


Table 8.4: An XPLAN Object.

8.3.4 Partially Executed Plans

The (partially and fully) executed programmes are represented as xplan. An example of an xplan object from degree planning is depicted in Table 8.4. In the table, Sid is the executed plan identifier, PEST (the plan execution start time) is the degree enrolment date, Results is a set of (course result temp-interval-course-completed) tuples.

8.3.5 Experiential Knowledge

We have seen in Chapter 4 how learned structures are derived from experiential language and represented as COBWEB clusters, LStruct. Case-Info is represented as a database of case objects. Table 8.5 depicts an instance of a case object.

In the table, Prog is the completed programme of studies, Categ-Order is the sequence of subjects undertaken in the programme ordered from highest to lowest weights, and Total-Wt is the aggregate weight of all the course weights in the programme.

8.4 Degree Evaluation and Verification

Both, plan evaluation and verification are part of the CDSP model. During the execution of a degree plan, evaluation of already executed regions of plans and verification of plans need to be done at appropriate times. A successful evaluation signals a successful completion of a plan and revision of unexecuted regions of a plan is required if contingencies arise (see §6.6.3 for background to plan verification and repair).
Case Object:

<table>
<thead>
<tr>
<th>Isa:</th>
<th>Case-Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Degree:</td>
<td>BSc</td>
</tr>
<tr>
<td>Categ-Order:</td>
<td>((Mathematics 10) (Computer-Science 9) (Physics 2))</td>
</tr>
<tr>
<td>Total-Wt:</td>
<td>21</td>
</tr>
</tbody>
</table>

Table 8.5: A Case Object.

The objective of degree evaluation is, given a partially completed programme of studies \textit{xplan} and a set of formal requirements, is to find answers to the questions "Does \textit{xplan} satisfy all the formal requirements? If not, why not?". An \textit{xplan} is evaluated against an existing set of formal constraints by checking it against the tester predicate for each of the constraints. The checking also involves checking the \textit{xplan} against some given result constraints \textit{result-cons} i.e., whether some specified results are attained for courses undertaken. (From Appendix C, the \textit{minimum level of performance rules} stipulate the requirements on the results.) In any academic degree domain, degree evaluation requires checking of performance in each course undertaken. In the ANU BSc domain, the following passing grades are: \textit{pass-grades} = \{Hd, D, Cr, P, P1, P2, CP, CRS\}, where Hd denotes high distinction, D denotes distinction, Cr denotes credit, P denotes pass, P1 denotes pass division 1, P2 denotes pass division 2, CP denotes conditional pass, and CRS denotes course requirements satisfied. There may be minimum performance criteria with respect to the pass-grades, which have to be met for progression to courses in later stages of a programme. A set of \textit{fail-grades} is also given but for the purposes of degree evaluation and degree planning, they are not relevant. Attainment of fail grades in courses does not allow the courses to be counted as part of a programme.

In [143], the difficulties confronted in automating a general degree evaluation task are discussed and a degree evaluator system for the ANU BSc degrees is presented.

During an execution of a degree plan, the plan has to be verified against the constraints and revised in light of any arising contingencies. The verification requires checking of the plan with an existing set of formal and preferential constraints, and in regards to any executed regions of the plan.
8.5 Degree Planning

Degree planning involves search for a plan of courses which is consistent with all the formal and preferential constraints. It involves consistently selecting and operating with a static set of formal constraints and a dynamic set of preferential constraints. Either all the constraints are known prior to problem solving and each new batch of constraints is processed anew (as realised in Model-Search), or the preferential constraints are issued one at a time and the plan developed incrementally (as realised in IPLAN). The former approach is suited to cases where the student knows all the preferential constraints prior to problem solving and does not want to explore any partial solution possibilities. The latter approach allows the student to interactively explore the solution space through preferential constraints. It requires reasoning with dynamic constraints as constraints usually change as interaction develops. Also, during the execution of degree plans, perturbations to existing plans result from revisions of preferential constraints and failing of courses in the plan. Both the approaches support revisions in events of contingencies.

The Model-Search and IPLAN approaches for degree planning are illustrated in Appendices D and E respectively.

8.6 Summary of Results

8.6.1 Constraints

Both the Model-Search and IPLAN problem solvers have been tested on BSc degree planning from ANU. They have been tested on 1989 BSc requirements (which form a set of formal constraints) and different sets of preferential constraints.

All the legislative requirements from years 1988-1993 for BSc degrees at the ANU are expressible in CL. Also, CL is rich enough to support declarations of typical preferential constraints in degree planning.

8.6.2 Model-Search

For each batch of preferential constraints and fixed formal constraints, Model-Search finds a programme of study if one exists. The constraint selection algorithm makes the choice of constraints for application informed—the most constraining inclusion constraints are applied in search prior to less constraining ones. Further, it has been found that use of consequent sets (i.e., sets of courses with prerequisites already in the partially developed programme), and clusters derived from case database as heuris-
tics significantly improve the performance of search. These heuristics are especially useful in guiding selection of conjunctive support structures for satisfying aggregate constraints. The former heuristic ensures that when selections for structures are made by default, those structures are chosen which have maximal supports already in the partially developed programme. In the latter, selection of those structures are forced which maximally share courses with an identified cluster of courses. This cluster of courses forms a super-set of courses in the partially developed programme.

In experimentations with Model-Search in degree planning, it has been found in many cases, that programmes satisfying a batch of preferential constraints can be drawn "off the shelf" from the case database. Model-Search allows "fine-tuning" of found programmes through addition and retraction of preferential constraints at runtime.

The scheduling component of degree planning is tightly constrained in that courses are available for undertaking in specified sessions—usually one course is offered in only programme one session. Any additional course availability constraints give rise to more flexibility in choices.

In many applications of Model-Search to degree planning, we have found that backtracking is not required. This results from the informed selections of conjunctive support structures in satisfaction of each constraint.

8.6.3 IPLAN

Given a set of formal constraints, IPLAN processes preferential constraints in the order they are received. IPLAN allows browsing of each new state and helps the user in formulation of the next constraint to apply. The development of a programme is incremental and transparent to the user. A partial programme from any state can be extended to completion automatically on instruction from the user.

Usually after processing all the preferential constraints, very few, if any, formal inclusion constraints are left for processing. Most of them get satisfied by the processing of preferential constraints.

Degree programme possibilities are explored one at a time through "what-if" analysis. Also, user-directed changes, including minimal changes, to a fully developed programme through preferential constraints are supported.
Chapter 9

Summary and Further Work

9.1 Introduction

In this chapter, we summarise the major results of this research and directions for further work are suggested.

9.2 CDSP Problems

In this thesis, we have identified a specialised but useful class of technically interesting real-world problems (called CDSP problems) for which existing problem solving techniques are inadequate and which forms a novel application of AI technology. The class of problems is formally abstracted and the characteristic properties (including mathematical properties) of the problems have been identified.

The CDSP problems are planning problems which require dynamic constraint satisfaction. The problem solving involves reasoning with a variety of constraints. There are two aspects to the problem solving: selection of plan objects, and their distribution into a linear sequence of bins such that all the constraints are satisfied. Some constraints naturally guide the selection process whilst the scheduling constraints specify restrictions on contents of the bins and length of the bin sequence. In addition to using constraints as Boolean expressions, they are used effectively in directing the search as an extension to the heuristic search paradigm. Domain heuristics, including experiential knowledge, are usually available in CDSP domains. They are also drawn upon and exploited in focusing the search.

Reasoning with the dynamically changing variety of constraints in CDSP class of problems makes it a rich and interesting domain for AI research. In these problems new constraints emerge during problem solving. In particular, the user explores solutions and directs search for a solution incrementally through additions and retractions.
of preferential constraints at run-time respecting the existing set of formal constraints. The CDSP problems require the following major features: detection of inconsistencies arising during search and their resolution; exploitation of available information in making search efficient; and the essential human dimension for the exploration of solutions and their incremental development.

9.3 CDSP Model

The inadequacy of existing technologies in solving the CDSP problems is argued and a model (called the CDSP model) is developed in light of these inadequacies. The model captures the pragmatic concerns in CDSP problem solving and is presented with its theoretical underpinnings. We have characterised the class of problems formally by giving the problem and solution statements and appropriate problem solving schemes. The computational architecture for the class of problems is presented in terms of a problem solving language called CL-IPS. CL-IPS coherently synthesises the following components of the model:

- a declarative input language called CL, a constraint language in which a CDSP problem is specified in terms of constraints,
- a hybrid knowledge representation language ORL in which knowledge is internally represented, and
- an intelligent problem solver IPS which, depending on the problem input, organises and performs search (either automatic, or user-guided and incremental) for a solution.

All constraints are encoded and input as statements in the constraint language CL. The semantics of the constraints are given in terms of the constructs in the language. CL is a rich language and all problems which conform to the CDSP defining characteristics are mappable to it. A repertoire of methods is provided for interpreting the constraints in CL.

Knowledge is represented in the CDSP model in a hybrid knowledge representation language called ORL which draws upon existing technology in knowledge representation. This representation allows packaging of related knowledge in the same environment.

The IPS component is the heart of the computational model. The computational procedures in IPS are developed for two formulations of CDSP problems. Both of them
Chapter 9. Summary and Further Work

draw heuristics from learned clusters of experiential knowledge. They are summarised later, after we summarise the general features of the CDSP model.

9.4 Constraints in Problem Solving

In CDSP problems there are typically a wide variety of constraints. They are functionally discriminated and used both as boolean expressions and as generators in search. The latter use is possible for inclusion constraints. It is a novel extension to the heuristic-guided search paradigm. The constraints are used in different stages of the search based on their functional groupings. For example, exclusion constraints are applied in pre-processing (i.e., prior to running the search). Inclusion constraints are used during search—for the selection of plan objects and their supports such that the respective inclusion constraint evaluates to true in the resulting point in search.

Partial ordering is defined on CDSP constraints in terms of constructs in the CL language. In CDSP problem solving, the ordering is important because constraints may get selected automatically for application at various stages of the search. The constraints are automatically selected by a selection algorithm which uses the ordering.

9.5 Heuristics

In CDSP problems, there is usually a database of previous cases. The COBWEB algorithm is incorporated as a component in the problem solving model. It derives "islands" (or clusters) of plan objects from the database based on maximal similarities and least differences in the cases. The derived information is drawn upon by the problem solver as heuristics for guiding search and for advising. The use of these heuristics effectively prunes the search space guiding selection of plan objects which are known to be part of previously successful plans.

9.6 Nogoods

In the computational schemes developed for CDSP problem solving, a database of nogoods is kept during problem solving. This database caches tuples of sets of values and a set of constraint labels. The values in a tuple are known to violate the constraints. These nogoods are exploited in guiding search away from points in the search space that are known to fail. This representation of constraints in the database allows constraints to be retracted and the nogoods database appropriately updated. Through constraint retraction inconsistencies recorded may also be removed.
9.7 Dynamic Constraint Satisfaction

Since CDSP problems are essentially dynamic constraint satisfaction problems, the problem solving model allows for dynamically changing constraints. Possible ways of satisfying an inclusion constraint is defined as the domain for the constraint with respect to the point in search, and a variable associated with the constraint is created and assigned one value. This value may later be revised if a solution cannot be found for the assignment.

Thus, the traditional requirement that all constraints, variables and values have to be known prior to search is not met by CDSP problems and is not a requirement in the CDSP model.

9.8 CDSP Problem Formulations

We have formulated CDSP problems in two ways in light of characteristically two different problem solving behaviors that are required in CDSP domains. Prior to summarising the two formulations and their associated computational procedures, we give below the problem solving structures adopted in both the schemes and the important knowledge structures which are central to both the schemes.

In both the formulations, there are two orthogonal components in CDSP problem solving: selection of plan objects along with their supports; scheduling of the selected objects (i.e., distribution of plan objects into a sequence of bins).

The plan objects are the objects (or activities) that are planned in CDSP problem solving. They have support structures which are central to both aspects of problem solving. These structures are AND/OR structures which capture support relations between plan objects and constraints. A property of a CDSP solution is that all the plan objects in the solution are sufficiently supported. Conjunctive support structures have the property that they contain plan objects and their single conjunctive supports. They are derived from the AND/OR structures and used in search, allowing the plan objects to be well-supported. They are also used to guide future selections of conjunctive support structures such that only single conjunctive support structures for plan objects find their way into solutions (i.e., in a solution, the unique support property of plan objects is satisfied).

During search, inclusion constraints are used to select conjunctive support structures for plan objects which satisfy these constraints and do not violate other constraints. These structures are cumulatively built for each search path. In the selection,
one of the candidate structures is chosen such that when the chosen structure is added to the existing cumulative structure, it respects the unique support property. In addition, the selection is guided by the heuristic that the selected structure maximally overlaps with the existing cumulative structure. The cumulative structure at each state in search is tested for schedulability by finding a schedule, if one exists, for the objects in the structure. The schedule preserves the support relations between the objects as captured in the support structure and satisfies all the scheduling constraints.

Scheduling with precedence constraints is known to be computationally intractable. The cumulative structures explicitly capture the support information \textit{per se} and are used in scheduling. CDSP scheduling is viewed as a constraint satisfaction problem in which a cumulative structure, a set of constraints restricting the number of plan objects in the bins, and a set of temporal constraints, are given. The task is to search for a schedule, which may be partially elaborated, and satisfies all the scheduling constraints. In a pre-processing step to scheduling, sophisticated temporal structures are derived from the cumulative structures and used in ordering search such that the constraint satisfaction engine considers the most constraining plan objects prior to the less constraining ones. Use of these structures give efficiency to the search for constraint satisfaction without a loss of completeness.

CDSP problems are formulated as Model-Search and IPLAN problems. These formulations and their respective computational schemes are summarised below.

\section*{9.8.1 Model-Search Problem}

In the Model-Search formulation, the problem is defined as sets of formal and preferential constraints and a set of plan objects. The objective is to find an interpretation of the set of constraints that satisfies all the formal constraints and a maximal subset of the preferential constraints. The solution to this problem is a sequence of bins containing plan objects.

Model-Search uses the constraints in searching for a solution. It finds a solution, if there is one, for a given set of constraints. The search is state-based. Constraints are automatically selected and systematically used in the search. At each state in search, an optimal constraint is automatically chosen for application with respect to the state. Nogoods, heuristics, and problem properties are used in guiding the selection of a conjunctive support structure which satisfies the chosen constraint and which minimises backtracking. Backtracking is invoked when a solution is not found and states cannot be expanded further. The search results in failure if a solution is not found after
performing an exhaustive search. Model-Search also allows minimal revisions of found solutions at run time through run-time constraints.

In Model-Search, the goal state is a state in which all the global constraints and any constraints accumulated on the search path evaluate to true. It is not defined prior to start of search but gets defined during search when support structures containing constraints are chosen.

Model-Search organises its search by a heuristic-based constraint selection algorithm which determines an optimal constraint for application at a given point in search. It does not require a priori knowledge of all the constraints. A variable and its associated domain of conjunctive support structures are dynamically created on receipt of each inclusion constraint.

The strengths of Model-Search lie in its exploitation of information at its disposal in search, and its addressing of the dynamic nature of CDSP problem solving. However, it has a major drawback in that it treats each change in constraints as a new problem and solves the new problem anew. This restricts user exploration of solution space which is an essential feature in CDSP problem solving.

9.8.2 IPLAN Problem

There is an increasingly justified belief that real-world expert system tasks should be interactive and incremental. That is, the reasoning engine should be able to support co-operation and shared hypothetical reasoning between human and expert systems. These features facilitate user-guided and exploratory problem solving which are essential in many such problems. CDSP type problems characteristically require these features and are realised in the IPLAN formulation of the problems.

The IPLAN problem is an extension of the Model-Search problem. In this problem, a set of formal constraints is given and preferential constraints are received in a sequence from the user. As each preferential constraint is received, the solution is incrementally developed (i.e., from one partial solution to another until a solution is found), by applying constraints as they are received. The user examines the partially developed solution at each state, and formulates the next constraint to add. The user may also seek advice from the problem solver on future constraints to add. Each of the IPLAN problem solving states are Model-Search problems. If all the user constraints get exhausted without a solution being found, Model-Search is invoked on the latest IPLAN problem solving state to get a solution if one exists. Usually, in CDSP problems, the user guides an incremental development of partial solutions to a solution.
During the user-guided problem solving, if the constraints become inconsistent then no partial solution results. The culprit constraints are identified to the user and relaxation of the culprit preferential constraints is sought from the user. At any time during IPLAN problem solving, the user may retract preferential constraints—giving the nonmonotonic character to IPLAN problem solving. Retraction of culprit constraints results in the updating of the solution sequence by tracking the position of the application of the culprit constraint in the sequence and re-applying the constraints appearing after the culprit constraint to the partial solution before the tracked position.

Computationally, the requirements in IPLAN problem solving indicated the appropriateness of an ATMS-like computational tool. A single-context ATMS (called CDSP-ATMS) is suited to IPLAN problem solving—a multiple-context ATMS was unsuitable (and more of a hindrance than help in IPLAN problem solving). CDSP-ATMS is used in conjunction with dependency-directed backtracking (DDB).

The CDSP-ATMS is state-based and caches all assumptions chosen to satisfy the applied constraints in a state-based dependency (or justification) structure. Because each of these assumptions are conjunctive support structures (i.e., a structure of plan objects and constraints), inconsistencies may arise in default assumptions made by the problem solver. DDB tracks the causes of inconsistencies, jumps the problem solver to the appropriate state, revises the previously made choice and directs the re-start of problem solving from that state.

The CDSP-ATMS and DDB combination facilitates modeling of user behavior in exploratory and incremental problem solving. It succinctly caches the assumptions made to satisfy the constraints in the order they are received. The path through a CDSP-ATMS maps to an IPLAN solution structure. The dependency information captured in the CDSP-ATMS facilitates detection of causes of inconsistencies, which may arise during problem solving, through dependency-directed backtracking.

“What-if” analysis is also supported in the IPLAN problem solving. This enables the user to explore alternative partial solutions and is facilitated by the CDSP-ATMS and DDB combination.

9.9 Degree Planning

The degree planning problem motivated the current research work. The CDSP model is abstracted from the characteristics of the degree planning problem. In this thesis, planning BSc degrees from the Australian National University regulations is used to
9.10 Further Work

Research tends to raise as many questions as it answers. The current research work on CDSP model is no different. The following issues of interest have been identified for further research. Possible extensions to the current model in light of these issues have also been suggested in the following.

Theoretical Foundation

Although the CDSP problems have been formalised and mathematical formulations given, we need to theoretically capture the incremental and interactive problem solving behavior of IPLAN. This would give the necessary theoretical foundation for the model.

Changing Formal Constraints

As a simplifying assumption, in the current model, it is assumed that the sets of formal constraints defined at two different times do not contradict each other. Both the Model-Search and IPLAN problem solving schemes currently do not address this contingency. A plan developed at an earlier time with respect to formal constraints defined at that time, may become inconsistent with a set of formal constraints defined at a latter time. This concern is relevant if a plan is not fully executed. The current model needs to be enhanced so that it can reason with this contingency.

Scheduling

In the current model, the scheduling engine finds a schedule through an informed search for constraint satisfaction. If a schedule is not found for a cumulative structure, the search terminates with failure without blaming any values which violate the constraints nor the constraints that are violated. The scheduling process needs to be enhanced by adding an ATMS-type computational tool such that the culprits can easily be detected facilitating revisions of values responsible for failures. The use of the tool would also permit guidance of future selections of conjunctive support structures in the selection process such that the nogoods that are already known to result in unschedulability are avoided.
Model-Search Limitation

The Model-Search procedure, in its current form, just finds a solution, if one exists, for a set of constraints. Failing this, it does not attempt to find a solution which is consistent with formal constraints and a maximal subset of preferential constraints. This latter feature is an interesting research issue in the general constraint satisfaction area. Generation of all subsets of a given set of preferential constraints and checking the consistency of each of these subsets with a given set of formal constraints are intractable tasks. An efficient way to find a solution that satisfies a maximal set of preferential constraints will be a useful extension to the current research and indeed to the general constraint satisfaction research area.

Modeling User Behavior

The current attempt to model user behavior, as realised in IPLAN is simple. At each point in problem solving, the problem solver gives as feedback to the user, a partial solution in which all the preferential constraints from the preferential constraint sequence are applied. This partial solution is essentially from a spectrum of partial solutions with varying degrees of elaboration. In this spectrum, on the one extreme, all the constraints as unapplied constraints form a partial solution, and on the other extreme, all the constraints are applied and a Model-Search solution appears for the set of constraints. That partial solution from the spectrum needs to be given as a feedback which maximally helps the user in his/her formulation of next constraint. The problem solver has to decide on which partial solution from the spectrum of partial solutions is best suited as feedback to the user. The existing IPLAN problem solver thus needs to be enhanced with judgemental rules for deciding on which partial solution to choose for the feedback and further search.

Also, modeling of user intentions is an interesting and potential extension to the current user modeling by IPLAN.
9.10. Further Work

Expert System Shell

During this research, most of the kernel of the intended expert system shell has been developed for the CDSP class of problems. It needs to be enhanced by extending the problem solving language (i.e., the constraint language and the problem solving components), such that more complicated constraints and the above suggested extensions are incorporated into the shell. Also, a user-interface needs to be built that supports user communication with the system. Eventually, it is expected that the current work will mature into an expert system shell for a wide range of CDSP type problems.
Bibliography

Bibliography


Appendix A

COBWEB Classification in Degree Planning Domain

In the following, we present the COBWEB parameters, describe the attributes in the degree planning data, describe data input to COBWEB and the classifications produced by COBWEB. We give an example of data input to COBWEB and results obtained from it.

A.1 COBWEB’s Parameters

When the predictiveness score (out of 1 is greater than or equal to this parameter, the feature is considered predictive of the node.

**Predictive-At = 0.89**

When the confidence score (out of 1 is greater than or equal to this parameter, the feature is considered to be shared by all instances and subgeneralisations stored at the node.

**Confident-At = 0.2**

Cutoff for when the evaluation function is not different enough to create a new class.

**Cutoff-New-Class = 2.0**

Cutoff for when the evaluation function is not different enough to extend the hierarchy one level down by creating new terminal nodes.

**Cutoff-New-Terminals = 2.0**

A.2 Attribute Description for Degree Planning Data

The instances in the degree planning data have names, which are numbers. Instance-number is the first attribute. The attributes for the instances are defined as follows.
Appendix A. COBWEB Classification in Degree Planning Domain

ACC1001

<table>
<thead>
<tr>
<th>Type</th>
<th>Nominal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Included</td>
<td>t</td>
</tr>
<tr>
<td>Possible-Values</td>
<td>(0 1)</td>
</tr>
<tr>
<td>Average-Calculator</td>
<td>(\lambda (x y) (if (equal x y) x 0))</td>
</tr>
<tr>
<td>Distance-Calculator</td>
<td>(\lambda (x y (if (equal x y) 0 1))</td>
</tr>
<tr>
<td>Documentation</td>
<td>“A Subject named ACC1001&quot;</td>
</tr>
</tbody>
</table>

END

ACC2001

<table>
<thead>
<tr>
<th>Type</th>
<th>Nominal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Included</td>
<td>t</td>
</tr>
<tr>
<td>Possible-Values</td>
<td>(0 1)</td>
</tr>
<tr>
<td>Average-Calculator</td>
<td>(\lambda (x y) (if (equal x y) x 0))</td>
</tr>
<tr>
<td>Distance-Calculator</td>
<td>(\lambda (x y (if (equal x y) 0 1))</td>
</tr>
<tr>
<td>Documentation</td>
<td>“A Subject named ACC2001&quot;</td>
</tr>
</tbody>
</table>

END

A.3 Data Input

Cases from results data base are input to the CDSP-COBWEB system in the following format. Fifty cases have been selected at random from the data base of ANU BSc results from the interval 1982-1988 for the purposes of illustration. The instance number precedes each successful case. The cases are sets of units.

Instance Successful BSc Cases


45 COMP1003 MATH1002 CHEM1002 ZOOL1002 PSYC1001 BIOC2001 BOT2004 MBI02003 BIOC2004 BOT3005 MBI02004 UNSP9012 BIOC3005 PSYC2007

A.3. Data Input

68 COMP1003 MATH1011 COMP1004 MATH1012 CHEM1002 ZOOL1002 BIOC2001
BIOC3005 BIOC3007 MBIO2003 BIOC3002 BIOC3008

88 STAT1003 STAT1004 BOT1001 CHEM1002 ZOOL1001 ZOOL2010 BOT2006
BIOC2001 ZOOL2011 ZOOL2012 LAW1101 BOT3002 ZOOL3013 ZOOL3015
ZOOL3016 ZOOL3014 BOT3010 ZOOL3018

95 MATH1011 COMP1001 MATH1012 COMP1002 CHEM1001 PHYS1001
BIOC2004 BIOC3005 CHEM3001 CHEM3003 BIOC3002 CHEM3002

110 STAT1003 STAT1004 PSYC1001 ANTH1001 POPS2001 MATH1001 PSYC2006
PSYC2001 PSYC2005 PSYC2002 POPS2002 MATH1002 PSYC2009 PSYC3001
PSYC3007 STAT2001 COMP1003 PSYC3008 PSYC3010

113 STAT1003 STAT1004 BOT1001 ZOOL1001 CHEM1002 BIOC2001 BOT2004
MBIO2003 BIOC2004 MBIO2004 BOT3005 BIOC3005 BIOC3007 BOT3011
BIOC3002

120 STAT1003 MATH1002 BOT1001 CHEM1002 ZOOL1001 BIOC2001 MBIO2003
ZOOL2010 MBIO2004 ZOOL2011 BIOC2004 BIOC3007 BIOC3005 ZOOL3033
BIOC3008

123 44015 43795 44004 44628 40653 40017 43795 44316 44628 44004 44156 40017
40482 44048 20008 4349 40084 20309 40653 MATH2011 MATH2013 STAT2001
STAT2004 FREN2001 MATH3001 MATH3005 MATH3011 MATH3013 STAT2001
MATH3016 MATH3014 MATH3002 MATH3012 MATH2012 MATH3004 PSYC1001
MATH3003 STAT3003 PSYC1001 MATH1024

128 STAT1003 MATH1011 STAT1004 MATH1012 PSYC1001 SOC1001 COMP1001
STAT3003 PSYC3008 MATH2011 PSYC3001 STAT3009 PSYC3010

135 COMP1003 COMP1005 CHEM1001 GEOL1001 ZOOL1001 GEOL2001
GEOL3005 GEOL3001
Appendix A. COBWEB Classification in Degree Planning Domain


147 44048 43494 44349 44004 20343 40482 20741 40084 COMP2011 MATH2001 MATH2003 COMP2012 MATH2002 COMP2013 COMP3012 MATH3003 MATH3005 MATH2004 COMP2013 MATH2013 MATH3001 STAT3003 COMP2012 MATH3004 MATH3002 MATH3005 MATH3001 STAT3001


157 UNSP9060 COMP1001 COMP1002 MATH2023 COMP2011 COMP2012 COMP2013 COMP3017 COMP3036 COMP3031


184 ZOOL2012 BOT1001 CHEM1002 UNSP9002 ZOOL1001 UNSP9011 UNSP9011 ZOOL2010 ZOOL2011 ZOOL3013 ZOOL3015 BOT2004 ZOOL3014 ZOOL3018 BOT3005


A.3. Data Input


201 MATH1001 STAT1003 CHEM1002 ZOOL1002 PSYC1001 BIOC2001 MBIO2003 PSYC2001 BIOC2004 MBIO2004 BIOC3005 BIOC3007 BIOC3002


254 20837 16777 40233 15844 40017 19961 40835 19262 40017 44350 40835 GEOL2003 GEOL2004 STAT1003 CHEM1001 STAT1003 GEOL3002 GEOL3004 GEOL3003 GEOL3001 MATH1001 GEOL3008 COMP1003

Appendix A. COBWEB Classification in Degree Planning Domain

STAT2001 MATH2003 STAT2002 MATH2004 COMP1005 PHIL2052 JAP1001 MATH2011 STAT3003 MATH3001 MATH3005 STAT3001 STAT2002 MATH3002 MATH2012 MATH3016 MATH3004 PHIL2052 MATH2013 MATH3003 MATH3011 STAT3003


265 MATH1001 MATH1002 CHEM1002 PSYC1001 ZOOL1001 BIOC2001 MBIO2003 ZOOL2010 BIOC2004 MBIO2004 ZOOL2011 PREH1001 BIOC3005 ZOOL3031 ZOOL3033 BIOC3002 ZOOL3034


313 STAT1003 STAT1004 CHEM1002 PSYC1001 ECON1001 PSYC2001 MATH1001 PSYC2007 MATH1002 PSYC2002 PSYC2008 PSYC2009 PSYC3001 PSYC3008 PSYC3002 PSYC3010


328 MATH1011 MATH1012 CHEM1001 PHYS1001 ZOOL1001 BIOC2001 BOT2004 MATH2011 MATH2013 BIOC2004 BOT3005 MATH2014 MATH2012 BIOC3005 BIOC3007 MATH3011 MATH3001 BIOC3002 BIOC3008

331 UNSP9060 PSYC2001 PSYC2007 PSYC2008 PSYC3010 PSYC2009 PSYC3001 PSYC3015 STAT1003

350 MATH1002 STAT1003 CHI1003 MATH1001 STAT1003 BOT1001 ZOOL1001 GEOG1002 BOT2004 ZOOL2021 BOT2006 BOT3005 GEOG2002 PHIL1002 PHIL2064 GEOG3006 PHIL2068 PHIL2067
A.3. Data Input

352 UNSP9060 ZOOL1002 PSYC1001 PSYC2007 PSYC2002 PSYC3001 PSYC3015 PSYC3010

360 STAT1003 STAT1003 CHEM1002 BOT1001 ZOOL1002 MATH1011 STAT1004 BOT1001 CHEM1002 ZOOL1001 BIOC2001 MBIO2003 BOT2004 BIOC2004 MBIO2004 CHEM2007 BIOC3005 BIOC3007 BOT3011 MATH1001 BIOC3002 BIOC3008 MBIO2004 ZOOL3035 BIOC3007 BOT3011

368 MATH1001 STAT1003 PSYC2007 PSYC2098 PSYC3099 PSYC3099 PSYC2008 PSYC2002 PSYC1001 ZOOL1002 2082 PSYC2001 PSYC3014 PSYC3008


407 STAT1003 MATH1002 CHEM1002 ZOOL1002 PSYC1001 PSYC2007 ZOOL2021 BOT2004 PSYC2002 ZOOL2023 BOT3005 ZOOL3033 PSYC3001 PSYC3015 PSYC3010 PSYC3016 ZOOL3036
CDSP-COBWEB transforms this data into instances of all possible units contained in the cases. The instances are sequences with 0's and 1's for corresponding unit presence or absence in each case.

For example, the sequence used for the degree planning case data is:

A.3. Data Input

GEOG3002 GEOG3006 GEOG3009 GEOG4001 GEOL1001 GEOL2001 GEOL2002
GEOL2003 GEOL2004 GEOL2005 GEOL3001 GEOL3002 GEOL3003 GEOL3004 GEOL3005
GEOL3006 GEOL3007 GEOL3008 GEOL3010 GEOL3011 GEOL3013 GEOL4001 GEOL4501
GEOL4502 GEPR3001 GER1001 GER1002 GER1003 GER1004 GER2003 GER2021
JAP1001 LAW1001 LAW1101 LAW2102 LING1001 LING1004 MATH1001 MATH1002
MATH1011 MATH1012 MATH1021 MATH1022 MATH1023 MATH1024 MATH2001
MATH2018 MATH2021 MATH2022 MATH2023 MATH2024 MATH2026 MATH3001
MATH3002 MATH3003 MATH3004 MATH3005 MATH3011 MATH3012 MATH3013
MATH3014 MATH3016 MATH3018 MATH3021 MATH3022 MATH3023 MATH3024
MATH3025 MATH3026 MATH3027 MATH3028 MATH3029 MATH3031 MATH3032
MATH3033 MATH3034 MATH3035 MATH3036 MATH3037 MATH3038 MATH3041
MATH4005 MATH4006 MATH4007 MATH4008 MBIO2003 MBIO2004 MED1001 NEUR4001
PHIL1001 PHIL1002 PHIL2002 PHIL2030 PHIL2052 PHIL2055 PHIL2057 PHIL2060
PHIL2061 PHIL2064 PHIL2067 PHIL2068 PHIL2080 PHIL3021 PHIL3022 PHIL3053
PHIL3054 PHYS1001 PHYS2001 PHYS2003 PHYS2004 PHYS2006 PHYS2008 PHYS3001
PHYS3002 PHYS3011 PHYS3012 PHYS3014 PHYS3015 PHYS3016 PHYS3017 PHYS4001
PHYS4002 PHYS4003 POL1001 POL2009 POL2035 POL2039 POL2047 POPS2001
PSYC2009 PSYC2098 PSYC3001 PSYC3002 PSYC3005 PSYC3006 PSYC3007 PSYC3008
PSYC3009 PSYC3010 PSYC3013 PSYC3014 PSYC3015 PSYC3016 PSYC3099 PSYC4001
SEAB1001 SEAB2001 SEAV1001 SOC1001 SOC2006 SOC2007 SOC2008 SOC2018
SOC2019 SOC2022 SOC2027 STAT1001 STAT1003 STAT1004 STAT2001 STAT2002
STAT2004 STAT3001 STAT3002 STAT3003 STAT3008 STAT3009 STAT4001 SWA1001
SWAA2001 SWAH1001 SWAH3001 UNSP9001 UNSP9002 UNSP9003 UNSP9004 UNSP9005
UNSP9008 UNSP9011 UNSP9012 UNSP9021 UNSP9022 UNSP9026 UNSP9027 UNSP9029
ZOOL2010 ZOOL2011 ZOOL2012 ZOOL2021 ZOOL2022 ZOOL2023 ZOOL3013 ZOOL3014
ZOOL3015 ZOOL3016 ZOOL3017 ZOOL3018 ZOOL3019 ZOOL3031 ZOOL3032 ZOOL3033
ZOOL3034 ZOOL3035 ZOOL3036 ZOOL4001 ZOOL4002

The instances for the first two cases of the above input data as derived using the above sequence are:
A.4 Data Output

The CDSP-COBWEB system produces the following clustering.

OUTPUT FROM CDSP-COBWEB, ON : (35 25 11 1 5 1992 2 nil -10)

Level 0 : node1 (# instances = 50)
all units

Level 1 : node21 (# instances = 7)

Level 2 : node89 (# instances = 2)
A.4. Data Output

BOT1001 ZOOL1001 ZOOL2010 ZOOL2011

Level 1: node8 (# instances = 7)
MATH1011 MATH1012 MATH2001 MATH2002 MATH2003 MATH2011
MATH2012 MATH2013 MATH2014 MATH3001 MATH3005 STAT2001
STAT2002 STAT3003 STAT3009

Level 2: node282 (# instances = 3)
COMP1001 COMP1005 MATH1023 MATH2004 MATH3004 MATH3011
MATH3012 MATH3016 STAT1003 STAT1004 STAT2004 STAT3001
SWAA1001

Level 1: node36 (# instances
PSYC1001 PSYC2002 PSYC2007 PSYC2009 PSYC3001 PSYC3008
PSYC3010 STAT1003

Level 2: node79 (# instances = 4)
MATH1001 MATH1002 PSYC2001 PSYC2008 STAT1004

Level 2: node80 (# instances = 2)
STAT1004 STAT2001 STAT2002

Level 1: node70 (# instances = 5)
MATH2013 MATH3001 MATH3003 MATH3004 MATH3005 MATH3011
MATH3016 STAT3003 40084 40482 44048

Level 2: node119 (# instances = 2)
MATH3002 STAT2004 STAT3001 44349

Level 2: node145 (# instances = 4)
COMP2011 COMP2012 COMP2013 COMP3013 COMP3015 COMP3017
COMP3031 40385 44048 44349

Level 2: node29 (# instances = 2)
Appendix A. COBWEB Classification in Degree Planning Domain

BIOSC2001 BIOC2004 CHEM2005 CHEM2006 COMP1001 COMP1002
PHYS1001

Level 1: node3 (# instances = 14)
BIOSC2001 BIOC2004 BIOC3002 BIOC3005 BIOC3007 BIOC3008
BOT3005 CHEM1002 MBIO2003 MBIO2004 ZOOL1001

Level 2: node15 (# instances = 2)
BOT2004

Level 2: node396 (# instances = 2)
BOT2004

Level 2: node48 (# instances = 4)
STAT1003

Level 3: node61 (# instances = 2)
BOT2004

Total number of features 546. Nominal 546. Numeric 0.

** PARAMETERS. **
insts-have-name t
predictive-at 0.89
confident-at 0.60
cutoff-new-class 2.00
cutoff-new-terminals 2.00

** INSTANCES. **
Total number of instances 50.
Number of instances with missing values 0.
Maximum percentage of missing values 0.00 %.

** TIMING. **
Run time : 261.65 seconds.
Appendix B

Syntax of CL Input Language

In this appendix, we present the complete syntax of the CL language using an extended Backus-Naur form (EBNF). Table B.1 lists the EBNF symbols and their meanings. The syntax includes constructs for expressing constraints in CDSP class of problems, and also for meta-level control in CDSP problem solving. First, we present the domains on which the language is defined.

B.1 Domains

In the following subsections, we present the domains, knowledge structures and functions which are required in defining the semantics of the CL problem input language.

B.1.1 Standard Domains

- Boolean = \{true, false\}

- The set of positive real numbers, $\mathbb{R}^+$. 

- The set of positive integers, $\mathbb{N}$.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;...&gt;</td>
<td>meta variable</td>
</tr>
<tr>
<td>::=</td>
<td>replacement</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>{...}+</td>
<td>one or more arbitrary occurrences</td>
</tr>
<tr>
<td>[...]</td>
<td>options</td>
</tr>
</tbody>
</table>

Table B.1: EBNF Symbols
Appendix B. Syntax of CL Input Language

- Rel-Op = \{EQUAL-TO, AT-LEAST, AT-MOST\} with the corresponding semantic domain \{=, \geq, \leq\}.
- Deg-Pref = \{LEAST, LESS, MOST\}.

B.1.2 Problem-Specific Domains

- A finite set of Set-Names.
- A finite set of Set-Values.
- A finite set of constraints, \(CONST\), expressed in CL, with disjoint and covering subsets: formal constraints (\(FC\)), preferential constraints (\(Pref-Cons\)), and committed constraints (\(Com-Cons\)).
- A finite set of plan objects, called Plan-Objs, identified by elements from a set of unique identifiers (Obj-Id).
- A finite set of unique identifiers, called Plan-Id, to identify plans.
- A finite set of well-formed logic formulas expressing supports for objects (Obj-Supports).
- A finite set of positive real numbers, given as Obj-Wts, enumerating all possible weights for objects.
- A finite set of other plan object attributes, \(AT = \{AT_1, AT_2, \ldots, AT_i, \ldots, AT_n\}\).
- A finite set of sets of possible attribute values for each attribute, \(ATV = \{ATV_1, ATV_2, \ldots, ATV_i, \ldots, ATV_n\}\).
- A finite set of temporal intervals (\(Temp-Int\)) in discrete time units in which objects are available.
- A finite set of possible plan completion times (\(Temp-Class\)) in some pre-defined discrete time units. Each element of this set is a name for a sequence of temporal intervals from \(Temp-Int\).
- A set of plan objects is given as the committed constraints (\(Com-Cons\)). This set of objects is a fixed part of all plans that are searched.
- A structure of clusters derived from experiential knowledge (LStruct).
• Various pieces of statistical information which are derived from experiential knowledge are captured in a structure called Case-Info.

In this thesis, Case-Info structures were defined for the degree planning problem but a scheme for their computation in arbitrary CDSP domains has not been developed. The constraint language needs to be enhanced such that attributes for statistical information derivation can be expressed in it and the structures appropriately derived.

B.2 CL Syntax

B.2.1 Problem Statement in CL

\[ \text{<problem> ::= <kb-declaration><ips> } \]
\[ <\text{ips}> ::= \text{KMS | UPDATE-CASE-DATABASE | DERIVE-CASE-CLUSTERS |} \]
\[ \text{EVALUATE-PLAN <plan-id> | VERIFY-PLAN <plan-id> | Model-Search | IPLAN | SAVE-PLAN} \]
\[ <\text{kb-declaration}> ::= [ <\text{file-dec}> ] <\text{bk-declaration}> \{ <\text{Pref-Cons}> \}^+ \]

B.2.2 Background Knowledge Declaration

\[ <\text{file-dec}> ::= <\text{background-db-file-name}> \]
\[ <\text{plan-db-file-name}> <\text{xplan-db-file-name}> \]
\[ <\text{bk-declaration}> ::= <\text{set-decs}> <\text{temp-decs}> <\text{plan-objs}> \{ <\text{Formal-Cons}> \}^+ \]
\[ [ <\text{Case-Cons}> ] [ <\text{Com-Cons}> ] \]
\[ <\text{set-decs}> ::= \{ <\text{set-name}> \text{SET-OF} <\text{set-values}> \}^+ \]
\[ [ [ <\text{set-name}> \text{INC} <\text{set-name}> ]^+ ] \]
\[ [ [ <\text{set-name}> \text{INCL} <\text{set-name}> ]^+ ] \]
\[ <\text{temp-decs}> ::= <\text{temporal-interval-sequence}> [ <\text{named-temp-interval-combinations}> ] \]
\[ <\text{plan-objs}> ::= \{ <\text{plan-object}> \{ <\text{obj-attrib.name}> <\text{obj-attrib.val}> \}^+ \}^+ \]

B.2.3 Formal Constraints

\[ <\text{Formal-Cons}> ::= <\text{weight-requirements}> | <\text{object-constraints}> | \]
\[ <\text{temporal-constraints}> | <\text{incompat-objs}> \]
\[ <\text{load-constraints}> \]
\[ <\text{weight-requirements}> ::= \text{SUM-WTS} <\text{class}> <\text{rel-op}> <\text{pos-real-number}> \]
\[ | \text{N-OBJS} <\text{class}> <\text{rel-op}> <\text{pos-integer}> \]
\[ <\text{class}> ::= \text{ALL} | \{ <\text{obj-attrib.val}> \}^+ \]
Appendix B. Syntax of CL Input Language

\[ \langle \text{object-temp-cons} \rangle ::= \langle \text{temporal-interval} \rangle \]

\[ \langle \text{object-constraints} \rangle ::= \langle \text{plan-object} \rangle \langle \text{obj-wt} \rangle \langle \text{obj-avail} \rangle \]

\[ \langle \text{obj-supports} \rangle ::= \langle \text{incomp-objs} \rangle \langle \text{co-objs} \rangle \]

\[ \langle \text{incomp-objs} \rangle ::= \text{INCOMP} \langle \text{plan-object} \rangle \langle \text{plan-objects} \rangle \]

\[ \langle \text{incompat-objs} \rangle ::= \text{N-OF} \langle \text{pos-integer} \rangle \langle \text{plan-objects} \rangle \]

\[ \langle \text{plan-objects} \rangle ::= \{ \langle \text{plan-object} \} \}^+ \]

\[ \langle \text{plan-object} \rangle ::= \langle \text{plan-object.id} \rangle \]

\[ \langle \text{co-objs} \rangle ::= \text{COREQ} \langle \text{plan-object} \rangle \langle \text{plan-objects} \rangle \]

\[ \langle \text{obj-wt} \rangle ::= \text{WT} \langle \text{plan-object} \rangle \langle \text{pos-real-number} \rangle \]

\[ \langle \text{obj-avail} \rangle ::= \text{OBJ-AVAIL} \langle \text{plan-object} \rangle \{ \langle \text{temporal-interval} \}^+ \]

\[ \langle \text{obj-supports} \rangle ::= \text{PREQ} \langle \text{plan-object} \rangle \langle \text{condition} \rangle \]

\[ \langle \text{condition} \rangle ::= \text{INCLUDE} \langle \text{plan-object} \rangle | \langle \text{weight-requirements} \rangle | \]

\[ \langle \text{temporal-constraints} \rangle ::= \langle \text{temporal-class} \rangle \langle \text{rel-op} \rangle \langle \text{pos-real-number} \rangle \]

\[ \langle \text{load-constraints} \rangle ::= \langle \text{temporal-interval} \rangle \langle \text{rel-op} \rangle \langle \text{pos-real-number} \rangle \]

\[ \langle \text{obj-attrib.name} \rangle ::= \text{An attribute name from the set AT} \]

\[ \langle \text{obj-attrib.val} \rangle ::= \text{An attribute value from the set ATV} \]

\[ \langle \text{pos-real-number} \rangle ::= \text{A positive real number from \( \mathbb{R}^+ \)} \]

\[ \langle \text{pos-integer} \rangle ::= \text{A positive integer from \( \mathbb{N} \)} \]

B.2.4 Preferential Constraints

\[ \langle \text{Pref-Cons} \rangle ::= \langle \text{inclusion} \rangle | \langle \text{exclusion} \rangle | \]

\[ \langle \text{temp-cons} \rangle | \langle \text{load-cons} \rangle | \langle \text{change-cons} \rangle \]

\[ \langle \text{inclusion} \rangle ::= \text{INCLUDE} \langle \text{plan-object} \rangle \{ \langle \text{deg-pref} \rangle \} \]

\[ \sum \langle \text{wts} \rangle \langle \text{class} \rangle \langle \text{rel-op} \rangle \langle \text{pos-real-number} \rangle \]

\[ \langle \text{change-cons} \rangle ::= \langle \text{temp-cons} \rangle | \langle \text{load-cons} \rangle | \langle \text{change-cons} \rangle \]
\[ \text{N-OBJ} <\text{class}><rel-op><pos-integer> } \\
<\text{exclusion}> ::= \text{EXCLUDE} <\text{plan-objects}> \mid \text{NOT-TAKE-OBJ-FROM} <\text{class}> \\
<\text{plan-objects}> ::= \{<\text{plan-object}>\}^+ \\
<\text{plan-object}> ::= <\text{plan-object.id}> \\
<\text{temp-cons}> ::= \text{TAKE-IN} <\text{plan-objects}> <\text{temporal-interval}> \\
& \text{TAKE-CONC} <\text{plan-objects}> \\
& \text{TAKE-BEFORE} <\text{plan-object}><\text{plan-object}> \\
& \text{PLAN-TIME} <\text{pos-integer}> <\text{temporal-class}> \mid \\
& \text{PLAN-TIME-BETWEEN} <\text{pos-integer}> <\text{temporal-class}> \\
\text{AND} <\text{pos-integer}> <\text{temporal-class}> \\
<\text{deg-pref}> ::= \text{MOST} \mid \text{LESS} \mid \text{LEAST} \\
<\text{change-cons}> ::= \text{FIND-ANOTHER-SOLN} \mid \text{SWAP}<\text{plan-object}><\text{plan-object}> \\
& \text{KEEP-ALL-BUT}<\text{plan-objects}> \mid \text{CHANGE-CATEG-PREF} \\
\]

**B.2.5 Committed Constraints**

\[ <\text{Com-Cons}> ::= <\text{xplan}> \]

Table B.2 summarises the CL vocabulary.

**B.2.6 Meta-Level Constructs for Invoking Problem Solving**

There are four, not necessarily independent, components in the CDSP problem solving viz., a knowledge management system (called KMS), a plan verification system, an "informed" automatic search for a plan, if one exists, through model search (called Model-Search), and an interactive and incremental planner (called IPLAN). These are invoked by the KMS, Model-Search and IPLAN statements in CL respectively.

The background knowledge is managed user-interactively (through retrieval, lookup and update operations) by a knowledge management system called KMS. The KMS language is based on the existing object-oriented representation of knowledge (as in KEE) and is briefly reported in [143].

\text{UPDATE-CASE-DATABASE} updates an existing database of previous cases against some existing set of formal constraints. All the cases which are not consistent with the existing set of formal constraints are filtered.

\text{DERIVE-CASE-CLUSTERS} derives a hierarchy of sets of plan objects from previous cases using the COBWEB clustering algorithm (see §4.4.11, page 78). The clusters are formed based on differences and similarities between plan objects.
### Appendix B. Syntax of CL Input Language

<table>
<thead>
<tr>
<th><strong>Words</strong></th>
<th><strong>Meanings</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>KMS</td>
<td>instruction to invoke knowledge management component</td>
</tr>
<tr>
<td>UPDATE-CASE-DATABASE</td>
<td>instruction to update existing case database with respect to some existing constraints</td>
</tr>
<tr>
<td>DERIVE-CASE-CLUSTERS</td>
<td>instruction to invoke the learning module COBWEB to derive a hierarchy of clusters</td>
</tr>
<tr>
<td>EVALUATE-PLAN</td>
<td>instruction to check whether a partially completed plan is admissible</td>
</tr>
<tr>
<td>VERIFY-PLAN</td>
<td>instruction to verify some plan against formal constraints</td>
</tr>
<tr>
<td>Model-Search</td>
<td>instruction to invoke automatic search for one solution</td>
</tr>
<tr>
<td>IPLAN</td>
<td>instruction to invoke interactive search for one or more solutions</td>
</tr>
<tr>
<td>SUM-WTS</td>
<td>aggregation of weights over plan objects</td>
</tr>
<tr>
<td>WITH</td>
<td>with</td>
</tr>
<tr>
<td>EQUAL-TO</td>
<td>=</td>
</tr>
<tr>
<td>AT-LEAST</td>
<td>≥</td>
</tr>
<tr>
<td>AT-MOST</td>
<td>≤</td>
</tr>
<tr>
<td>N-OF</td>
<td>a positive integer n of</td>
</tr>
<tr>
<td>INCOREQ</td>
<td>object incompatibility</td>
</tr>
<tr>
<td>COREQ</td>
<td>co-requisite supports for an object</td>
</tr>
<tr>
<td>PREQ</td>
<td>precedence supports for an object</td>
</tr>
<tr>
<td>ALL</td>
<td>all plan objects in a plan</td>
</tr>
<tr>
<td>OR</td>
<td>logical “or”</td>
</tr>
<tr>
<td>AND</td>
<td>logical “and”</td>
</tr>
<tr>
<td>EXCLUDE</td>
<td>exclude</td>
</tr>
<tr>
<td>INCL</td>
<td>set inclusion (⊆)</td>
</tr>
<tr>
<td>INC</td>
<td>strict set inclusion (⊂)</td>
</tr>
<tr>
<td>INCLUDE</td>
<td>include</td>
</tr>
<tr>
<td>N-OBJ</td>
<td>n number of plan objects from</td>
</tr>
<tr>
<td>PLAN-TIME</td>
<td>total time for a plan</td>
</tr>
<tr>
<td>PLAN-TIME-BETWEEN</td>
<td>total time allowed for a plan between two limits</td>
</tr>
<tr>
<td>TAKE-OBJ-FROM</td>
<td>take plan objects belonging to a certain class</td>
</tr>
<tr>
<td>NOT-TAKE-OBJ-FROM</td>
<td>not take plan objects belonging to a certain class</td>
</tr>
<tr>
<td>TAKE-IN</td>
<td>take in</td>
</tr>
<tr>
<td>TAKE-CONC</td>
<td>take concurrently</td>
</tr>
<tr>
<td>TAKE-BEFORE</td>
<td>take before</td>
</tr>
<tr>
<td>FIND-ANOTHER-SOLN</td>
<td>find another solution</td>
</tr>
<tr>
<td>SWAP</td>
<td>swap a plan object with another</td>
</tr>
<tr>
<td>KEEP-ALL-BUT</td>
<td>keep all existing plan objects except the ones specified</td>
</tr>
<tr>
<td>CHANGE-CATEG-PREF</td>
<td>change general category preferences</td>
</tr>
<tr>
<td>WHAT-IF</td>
<td>“what-if” (conditional) queries</td>
</tr>
</tbody>
</table>

Table B.2: A Summary of the CL Vocabulary.
EVALUATE-PLAN \(<\text{plan-id}>\) checks whether some partially executed plan satisfies some current set of formal constraints. It advises on any violated constraints.

VERIFY-PLAN \(<\text{plan-id}>\) checks whether some previously accepted plan is currently admissible (i.e., whether any changes in formal constraints has made some previously consistent plan inconsistent). If it is not then the verifier returns \text{false}. Otherwise, it returns \text{true}.

Model-Search and IPLAN are the computational mechanisms developed for the CDSP class of problems. Model-Search is an automatic approach in which a plan is found, if one exists, for a given set of constraints. Any change to a given set of constraints results in a new problem and Model-Search solves it anew. The user browses the produced plans, issues constraints in CL and invokes Model-Search until he/she is satisfied with a plan. An informed backtrack search mechanism is employed in Model-Search. On the other hand, IPLAN is an interactive, incremental planner. The possibilities are filtered through preferential constraints and the plan structure gets elucidated progressively during problem solving. IPLAN makes use of Model-Search in finding a solution from an IPLAN partial solution once the user preferences get exhausted and an IPLAN solution is not fully developed.

All the problem solvers work from a background knowledge base file containing all the plan objects, formal constraints and global set declarations; a plan database file; and a database file of executed plans. They are identified to the system by \(<\text{background-db-file-name}>\), \(<\text{plan-db-file-name}>\) and \(<\text{xplan-db-file-name}>\) respectively.
Appendix C

ANU BSc Order

An extract of the 1989 ANU BSc (Units) Order is given in this appendix.

Citation

1A. This Order may be cited as the Degree of Bachelor of Science (Units) Order.

Interpretation

1. In this Order, unless the contrary intention appears-
   “degree” means the degree of Bachelor of Science;
   “Faculty” means the Faculty of Science;
   ...

Requirements for degree

2. (1) For the purposes of sub-rule 4(1) of the Rules, a candidate for the pass degree shall not be taken to have completed the course for the degree unless the candidate passes:

   (a) units offered by a science-related department that have a total value of at least 12 points; and

   (b) as many additional units as are required to make a total value of the units passed by the candidate not less than 20 points.

2. (2) The units referred to in subclause (1) include:

   (a) units selected from the units set out in Group A of the table that have a total value of at least 6 points;
(b) units selected from the units set out in Group B of the table that have a total value of at least 2 points;
(c) units selected from the units set out in Group C of the table that are offered by a science-related department and that have a total value of at least 2 points;
(d) as many additional units selected from the units set out in Group C of the table as are required to make the total value of the units selected from Group C at least 4 points; and
(e) as many additional units selected from the units set out in Group A, B, C, D or E of the table as are required to make the total value of the units passed by the candidate at least 20 points.

2. (2A) The units referred to in subclause (1) shall not include:

(a) first-year units selected from the units specified in Group E of the table that have a total value of more than 2 points;
(b) units selected from the units set out in Group A of the table and first-year units selected from the units specified in Group E of the table that have a combined total value of more than 10 points; or
(c) units selected from the units specified in Group E of the table that have a total value of more than 6 points.

2. (3) A candidate may not count for the purposes of a paragraph of sub-clause (1) a point that is counted for the purposes of another paragraph of that sub-clause notwithstanding that the unit in respect of which that point was obtained is a unit that is set out in more than 1 of the groups, or in more than 1 of the parts of a group, in the table.

The Table

3. (1) For the purposes of sub-rule 5(1) of the Rules, the units for the pass degree of Bachelor of Science are the units set out in column 2 of the table.

3. (2) The value, for the purposes of the Rules and this order, of each of the units referred to in sub-clause (1) is the number of points specified opposite the unit in column 3 of the table.
Conditions with respect to certain units

4. (1) A unit chosen from a faculty other than the Faculty of Science shall be taken in accordance with the courses of study (Degree of Bachelor of Arts) Rules, the Courses of Study (Degree of Bachelor of Arts (Asian Studies)) Rules, the Courses of Study (Degree of Bachelor of Economics) Rules or the Courses of Study (Degree of Bachelor of Laws) Rules, as the case requires.

4. (2) Except with the approval of the Faculty, a candidate shall not take a unit set out in Group C of the table unless the candidate has obtained at least 6 points in respect of the units set out in Group A of the table.

4. (3) The Faculty may determine that a candidate may not take a unit specified in the determination unless the candidate has passed, or takes concurrently, another unit specified in the determination.

4. (4) A candidate may not count for the degree points in respect of-

(a) Chemistry A02 if points in respect of Chemistry A01 are so counted;

(aa) Chemistry A11 if points in respect of Chemistry A01 or Chemistry A20 are so counted;

(b) Physics A11 if points in respect of Physics A12 are so counted;

...

Minimum level of performance

For the purposes or rule 8 of the Rules-

5. (a) a candidate cannot count towards the pass degree of Bachelor of Science-

(i) more than 2 points in respect of units from Group A in the table in which the candidate’s performance has been classified as “pass division 2”; or

(ii) more than 2 points in respect of units from Groups B, C and D in the table in which the candidate’s performance has been classified as “pass division 2”; and

5. (b) a candidate shall not be permitted to the pass degree of Bachelor of Science unless the candidate’s performance has been classified as at least “pass division 1” in units having a total value of at least 3 points from Group C in the table.
Appendix D

Illustration of Model-Search with Degree Planning

In this appendix, the use of Model-Search in the ANU BSc degree planning example is illustrated. The 1989 BSc Regulations from [2] (see Appendix C) are used. Retrieval of programmes from the database of previous cases requires simple pattern-matching and is omitted in the illustration.

D.1 Search Structure and Constraints

Central to problem solving in Model-Search is the state-based search structure. A state, 

\[ S_i = <AC, C, EX, X, CS, PC, schedule> \]

where AC is a set of accumulated constraints, C is the constraint chosen for application to \( S_i \), \( EX \) is the choice made on the set of plan objects from the application of \( S_{i-1}.C \) to \( S_i \), \( X \) is a conjunctive support structure derived for \( EX \), CS is the cumulative structure combining all the structures in \( S_0 \) to \( S_i \), PC is a sequence of processed inclusion constraints from \( S_0 \) to \( S_{i-1} \), and schedule is one possible schedule derivable from CS.

Given a general initial state \( S_0 \): \( AC = \text{(get-cons } S_0.CS) \); \( C = \text{Nil} \); \( EX = \text{Nil} \); \( X = \text{Nil} \); \( PC = () \); and some CS and schedule, which may be empty, the objective is to find a goal state \( S_g \) in which the CS can be schedulable and satisfies all the accumulated constraints AC in \( S_g \) and the existing set of all existing formal and any preferential constraints \( \Delta \). \( \Delta \) is defined globally. Note that, in Model-Search, new constraints arise through support knowledge during search. These constraints are called accumulated constraints and are cached in AC in each state.
SQ.CS may be unexecuted, partially executed or empty. It may already satisfy some constraints from $\Delta$ as recorded in $S_0.PC$. The goal state contains a fully elaborated schedule consistent with $\Delta$ and $S_g.AC$.

In Model-Search, all the global constraints $\Delta$ are known prior to problem solving. In the ANU BSc degree planning example,

$$\Delta = \left( (\text{Sum-Wts A At-least 6}) \ (\text{Sum-Wts (Year1) E At-most 2}) \right) \left( (\text{Sum-Wts (Year1) (A E) At-most 10}) \ (\text{Sum-Wts E At-most 6}) \right) \left( (\text{Sum-Wts A At-least 6}) \ (\text{Sum-Wts A P2 At-most 2}) \right) \left( (\text{Sum-Wts (B C D) P2 At-most 2}) \right) \left( (\text{Sum-Wts C At-least 3 With P1 Or-better}) \right) \left( (\text{Sum-Wts Science At-least 12}) \ (\text{Sum-Wts All At-least 20}) \right) \left( (\text{Sum-Wts B At-least 2}) \ (\text{Sum-Wts Science C At-least 2}) \right) \left( (\text{Sum-Wts C At-least 4}) \ (\text{Sum-Wts All At-least 20}) \right) \left( (\text{Sum-Wts Yr3S2 At-Most 3}) \ (\text{Sum-Wts Yr3S1 At-Most 3}) \right) \left( (\text{Sum-Wts Yr2S2 At-Most 3}) \ (\text{Sum-Wts Yr2S1 At-Most 3}) \right) \left( (\text{Sum-Wts Yr1S2 At-Most 4}) \ (\text{Sum-Wts Yr1S1 At-Most 4}) \right) \left( (\text{Sum-Wts Yr3 At-Most 6}) \ (\text{Sum-Wts Yr2 At-Most 6}) \right) \left( (\text{Sum-Wts Yr1 At-Most 8}) \right) \left( (\text{Sum-Wts All At-least 20}) \right) \right.$$ 

The following constraints from $\Delta$ are used as generator constraints. They are given here in order of applicability based on the constraint selection algorithm.

- (Sum-Wts Science C At-least 2)
- (Sum-Wts C At-least 4)
- (Sum-Wts B At-least 2)
- (Sum-Wts A At-least 6)
- (Sum-Wts Science At-least 12)
- (Sum-Wts Yr3S2 3)
- (Sum-Wts Yr3S1 3)
- (Sum-Wts Yr2S2 3)
- (Sum-Wts Yr2S1 3)
- (Sum-Wts Yr1S2 4)
- (Sum-Wts Yr1S1 4)
- (Sum-Wts All At-least 20)
**D.2 Execution of Model-Search for Degree Planning**

Here, we present a trace of *model-search* in ANU BSc degree planning. (Text in bold font denotes user input.)

>`(model-search)

Loading Knowledge Base......

....done!

Enter your student-id: **8910042**

/*Any existing plan for the student is displayed here.*/

Next Instruction?
1. Quit constraint addition/retraction
2. Quit planning
3. Add constraints
4. Display Formal Constraints

index> 1

>>Start-Trace<<

*SO AC: {}*

  *C: (Sum-Wts Science C At-least 2)*

  *EX: Nil*

  *X: Nil*

  *CS: Nil*

  *PC: Nil*

(*Note a plan is the schedule sequence reversed.*)
### Current Plan

| Year 1 Session 1 | Nil | (wt: 0) |
| Year 1 Session 2 | Nil | (wt: 0) |
| Year 2 Session 1 | Nil | (wt: 0) |
| Year 2 Session 2 | Nil | (wt: 0) |
| Year 3 Session 1 | Nil | (wt: 0) |
| Year 3 Session 2 | Nil | (wt: 0) |

\[ S1 \text{ AC: \{} \]
\[ C: \text{(Sum-Wts C At-least 4)} \]
\[ \text{EX: \{} \text{Bioc3002 Bioc3005} \]
\[ \text{X: \{} \text{(T Bioc3002) (Bioc3002 Bioc2001) (Bioc2001 Chem1001)} \]
\[ \text{ (T Bioc3005) (Bioc3005 Bioc2001}) \]
\[ \text{CS: \{} \text{(T Bioc3002) (T Bioc3005) (Bioc3002 Bioc2001) (Bioc2001 Chem1001) (Bioc3005 Bioc2001}) \]
\[ \text{PC: \{} \text{(Sum-Wts Science C At-least 2))} \]
### Current Plan

<table>
<thead>
<tr>
<th>Year 1 Session 1</th>
<th>Nil</th>
<th>(wt: 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year 1 Session 2</td>
<td>Nil</td>
<td>(wt: 0)</td>
</tr>
<tr>
<td>Over Year 1</td>
<td>Chem1001</td>
<td></td>
</tr>
<tr>
<td>Year 2 Session 1</td>
<td>Bioc2001</td>
<td>(wt: 1)</td>
</tr>
<tr>
<td>Year 2 Session 2</td>
<td>Nil</td>
<td>(wt: 0)</td>
</tr>
<tr>
<td>Year 3 Session 1</td>
<td>Bioc3005</td>
<td>(wt: 1)</td>
</tr>
<tr>
<td>Year 3 Session 2</td>
<td>Bioc3002</td>
<td>(wt: 1)</td>
</tr>
</tbody>
</table>

$S2$ AC: {}  

C: (Sum-Wts B At-least 2)  
EX: {Bioc3007 Bioc3008}  
(Bioc3005 Bioc2001))}  
CS: {((T Bioc3007) (T Bioc3008)  
(Bioc3007 Bioc2004) (Bioc2004 Bioc2001)  
(Bioc2001 Chem1001)  
(Bioc3008 Bioc2004)  
(T Bioc3002) (T Bioc3005) (Bioc3002 Bioc2001))}  
PC: ((Sum-Wts C At-least 4) (Sum-Wts Science C At-least 2))
### Appendix D. Illustration of Model-Search with Degree Planning

#### Current Plan

<table>
<thead>
<tr>
<th>Session</th>
<th>Course(s)</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year 1 Session 1</td>
<td>Nil</td>
<td>2</td>
</tr>
<tr>
<td>Year 1 Session 2</td>
<td>Nil</td>
<td>0</td>
</tr>
<tr>
<td>Over Year 1</td>
<td>Chem1001</td>
<td></td>
</tr>
<tr>
<td>Year 2 Session 1</td>
<td>Bioc2001</td>
<td>1</td>
</tr>
<tr>
<td>Year 2 Session 2</td>
<td>Bioc2004</td>
<td>1</td>
</tr>
<tr>
<td>Year 3 Session 1</td>
<td>Bioc3005 Bioc3007</td>
<td>2</td>
</tr>
<tr>
<td>Year 3 Session 2</td>
<td>Bioc3002 Bioc3008</td>
<td>2</td>
</tr>
</tbody>
</table>

**S3 AC:** \{

**C:** \((\text{Sum-Wts A At-least 6})\)

**EX:** Nil

**X:** Nil

**CS:** \{

\((T \text{ Bioc3007}) (T \text{ Bioc3008})\)

\((\text{Bioc3007 Bioc2004}) (\text{Bioc2004 Bioc2001})\)

\((\text{Bioc2001 Chem1001})\)

\((\text{Bioc3008 Bioc2004})\)

\((T \text{ Bioc3002}) (T \text{ Bioc3005}) (\text{Bioc3002 Bioc2001})\)\}

**PC:** \((\text{Sum-Wts B At-least 2}) (\text{Sum-Wts C At-least 4}) (\text{Sum-Wts Science C At-least 2})\)
### Current Plan

<table>
<thead>
<tr>
<th>Year</th>
<th>Session</th>
<th>Course</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year 1</td>
<td>Session 1</td>
<td>Nil</td>
<td>2</td>
</tr>
<tr>
<td>Year 1</td>
<td>Session 2</td>
<td>Nil</td>
<td>0</td>
</tr>
<tr>
<td>Over Year 1</td>
<td></td>
<td>Chem1001</td>
<td></td>
</tr>
<tr>
<td>Year 2</td>
<td>Session 1</td>
<td>Bioc2001</td>
<td>1</td>
</tr>
<tr>
<td>Year 2</td>
<td>Session 2</td>
<td>Bioc2004</td>
<td>1</td>
</tr>
<tr>
<td>Year 3</td>
<td>Session 1</td>
<td>Bioc3005, Bioc3007</td>
<td>2</td>
</tr>
<tr>
<td>Year 3</td>
<td>Session 2</td>
<td>Bioc3002, Bioc3008</td>
<td>2</td>
</tr>
</tbody>
</table>

\[S_4\ AC: \{}\]

\[C: (\text{Sum-Wts Yr3S2} \ 3)\]

\[EX: \{\text{Chem1002, Bot1001}\}\]

\[X: \{(T \text{Chem1002}) \ (T \text{Bot1001}) \ (\text{Chem1002}) \ (\text{Bot1001})\}\]

\[CS: \{(T \text{Chem1002}) \ (T \text{Bot1001}) \ (T \text{Bioc3007}) \ (T \text{Bioc3008}) \ (\text{Bioc3007} \ \text{Bioc2004}) \ (\text{Bioc2004} \ \text{Bioc2001}) \ (\text{Bioc2001} \ \text{Chem1001}) \ (\text{Bioc3008} \ \text{Bioc2004}) \ (T \ \text{Bioc3002}) \ (T \ \text{Bioc3005}) \ (\text{Bioc3002} \ \text{Bioc2001})\}\]

\[PC: ((\text{Sum-Wts A At-least} \ 6) \ (\text{Sum-Wts B At-least} \ 2) \ (\text{Sum-Wts C At-least} \ 4) \ (\text{Sum-Wts Science C At-least} \ 2))\]
<table>
<thead>
<tr>
<th>Current Plan</th>
</tr>
</thead>
</table>
| **Year 1 Session 1**: Nil  
  (wt: 3) |
| **Year 1 Session 2**: Nil  
  (wt: 3) |
| **Over Year 1**: Chem1001 Chem1002 Bot1001 |
| **Year 2 Session 1**: Bioc2001  
  (wt: 1) |
| **Year 2 Session 2**: Bioc2004  
  (wt: 1) |
| **Year 3 Session 1**: Bioc3005 Bioc3007  
  (wt: 2) |
| **Year 3 Session 2**: Bioc3002 Bioc3008  
  (wt: 2) |

$S_5$ AC: {}

C: (Sum-Wts Yr3S1 3)

EX: {Bot3005}

X: {{(T Bot3005) (Bot3005 Bot2004) (Bot2004 Bot1001)}}

CS: {{(T Bot3005) (Bot3005 Bot2004) (Bot2004 Bot1001)  
  (T Chem1002) (T Bioc3007) (T Bioc3008)  
  (Bioc3007 Bioc2004) (Bioc2004 Bioc2001)  
  (Bioc2001 Chem1001) (Bioc3008 Bioc2004)  
  (T Bioc3002) (T Bioc3005) (Bioc3002 Bioc2001)}}

PC: {{(Sum-Wts Yr3S2 3) (Sum-Wts A At-least 6)  
  (Sum-Wts B At-least 2) (Sum-Wts C At-least 4)  
  (Sum-Wts Science C At-least 2)}}
## D.2. Execution of Model-Search for Degree Planning

<table>
<thead>
<tr>
<th>Current Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year 1 Session 1 : Nil (wt: 3)</td>
</tr>
<tr>
<td>Year 1 Session 2 : Nil (wt: 3)</td>
</tr>
<tr>
<td>Over Year 1 : Chem1001 Chem1002 Bot1001</td>
</tr>
<tr>
<td>Year 2 Session 1 : Bioc2001 Bot2004 (wt: 2)</td>
</tr>
<tr>
<td>Year 2 Session 2 : Bioc2004 (wt: 1)</td>
</tr>
<tr>
<td>Year 3 Session 1 : Bioc3005 Bioc3007 (wt: 2)</td>
</tr>
<tr>
<td>Year 3 Session 2 : Bioc3002 Bioc3008 Bot3005 (wt: 3)</td>
</tr>
</tbody>
</table>

**S6 AC:** {
(Sum-Wts (Chemistry Physics Mathematics Statistics Computer-science) At-least 2))

**C:** (Sum-Wts Yr2S2 3)

**EX:** {Bot3009}

**X:** {

**CS:** {
(T Bot3005) (Bot3005 Bot2004) (Bot2004 Bot1001)
(T Chem1002) (T Bioc3007) (T Bioc3008)
(Bioc3007 Bioc2004) (Bioc2004 Bioc2001)
(Bioc2001 Chem1001) (Bioc3008 Bioc2004)
(T Bioc3002) (T Bioc3005) (Bioc3002 Bioc2001)

**PC:** {
(Sum-Wts Yr3S1 3) (Sum-Wts Yr3S2 3)
(Sum-Wts A At-least 6) (Sum-Wts B At-least 2)
(Sum-Wts C At-least 4) (Sum-Wts Science C At-least 2))

---

---
### Current Plan

<table>
<thead>
<tr>
<th>Year 1 Session 1</th>
<th>Nil</th>
<th>(wt: 3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year 1 Session 2</td>
<td>Nil</td>
<td>(wt: 3)</td>
</tr>
<tr>
<td>Over Year 1</td>
<td>Chem1001 Chem1002 Bot1001</td>
<td></td>
</tr>
<tr>
<td>Year 2 Session 1</td>
<td>Bioc2001 Bot2004</td>
<td>(wt: 2)</td>
</tr>
<tr>
<td>Year 2 Session 2</td>
<td>Bioc2004</td>
<td>(wt: 1)</td>
</tr>
<tr>
<td>Year 3 Session 1</td>
<td>Bioc3005 Bioc3007</td>
<td>(wt: 2)</td>
</tr>
<tr>
<td>Year 3 Session 2</td>
<td>Bioc3002 Bioc3008 Bot3005</td>
<td>(wt: 3)</td>
</tr>
</tbody>
</table>

$S7$ AC: \{$(\text{Sum-Wts}\ (\text{Chemistry}\ \text{Physics}\ \text{Mathematics}\ \text{Statistics}\ \text{Computer-science})\ \text{At-least}\ 2)\}$

C: $(\text{Sum-Wts}\ \text{Yr2S1}\ 3)$

EX: \{Phil3053\}

X: \{\{'T\ Phil3053\} (Phil3053 Phil2080)\}\\
CS: \{\{'T\ Phil3053\} (Phil3053 Phil2080)\\
    (T Bot3009) (Bot3009 Bot2002) (Bot2002 Bot1001)\\
    (Bot2002 (Sum-Wts\ (\text{Chemistry}\ \text{Physics}\ \text{Mathematics}\ \text{Statistics}\ \text{Computer-science})\ \text{At-least}\ 2))\\
    (T Bot3005) (Bot3005 Bot2004) (Bot2004 Bot1001)\\
    (T Chem1002) (T Bioc3007) (T Bioc3008)\\
    (Bioc3007 Bioc2004) (Bioc2004 Bioc2001)\\
    (Bioc2001 Chem1001) (Bioc3008 Bioc2004)\\
    (T Bioc3002) (T Bioc3005) (Bioc3002 Bioc2001)\}

PC: \{(\text{Sum-Wts}\ \text{Yr2S2}\ 3)\ (\text{Sum-Wts}\ \text{Yr3S1}\ 3)\ (\text{Sum-Wts}\ \text{Yr3S2}\ 3)\ (\text{Sum-Wts}\ \text{A}\ \text{At-least}\ 6)\ (\text{Sum-Wts}\ \text{B}\ \text{At-least}\ 2)\ (\text{Sum-Wts}\ \text{C}\ \text{At-least}\ 4)\ (\text{Sum-Wts}\ \text{Science}\ \text{C}\ \text{At-least}\ 2)\}
### Current Plan

<table>
<thead>
<tr>
<th>Year 1 Session 1</th>
<th>Nil (wt: 3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year 1 Session 2</td>
<td>Nil (wt: 3)</td>
</tr>
<tr>
<td>Over Year 1</td>
<td>Chem1001 Chem1002 Bot1001</td>
</tr>
<tr>
<td>Year 2 Session 1</td>
<td>Bioc2001 Bot2004 Phil2080 (Sum-Wts (Chemistry Physics Mathematics Statistics Computer-science) At-least 2)) (wt: 3)</td>
</tr>
<tr>
<td>Year 2 Session 2</td>
<td>Bioc2004 Bot2002 Phil3053 (wt: 3)</td>
</tr>
<tr>
<td>Year 3 Session 1</td>
<td>Bioc3005 Bioc3007 Bot3009 (wt: 3)</td>
</tr>
<tr>
<td>Year 3 Session 2</td>
<td>Bioc3002 Bioc3008 Bot3005 (wt: 3)</td>
</tr>
</tbody>
</table>

S8 AC: {(Sum-Wts (Chemistry Physics Mathematics Statistics Computer-science) At-least 2))}

C: (Sum-Wts Yr1S2 4)

EX: {}

X: {}

CS: {(T Phil3053) (Phil3053 Phil2080)
     (T Bot3009) (Bot3009 Bot2002) (Bot2002 Bot1001)
     (Bot2002 (Sum-Wts (Chemistry Physics Mathematics Statistics Computer-science) At-least 2))
     (T Bot3005) (Bot3005 Bot2004) (Bot2004 Bot1001)
     (T Chem1002) (T Bioc3007) (T Bioc3008)
     (Bioc3007 Bioc2004) (Bioc2004 Bioc2001)
     (Bioc2001 Chem1001) (Bioc3008 Bioc2004)
     (T Bioc3002) (T Bioc3005) (Bioc3002 Bioc2001)}

PC: ((Sum-Wts Yr2S1 3) (Sum-Wts Yr2S2 3) (Sum-Wts Yr3S1 3)
     (Sum-Wts Yr3S2 3)(Sum-Wts A At-least 6) (Sum-Wts B At-least 2)
     (Sum-Wts C At-least 4) (Sum-Wts Science C At-least 2))
### Appendix D. Illustration of Model-Search with Degree Planning

**Current Plan**

<table>
<thead>
<tr>
<th>Year 1 Session 1</th>
<th>Nil</th>
<th>(wt: 3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year 1 Session 2</td>
<td>Nil</td>
<td>(wt: 3)</td>
</tr>
<tr>
<td>Over Year 1</td>
<td>Chem1001 Chem1002 Bot1001</td>
<td></td>
</tr>
<tr>
<td>Year 2 Session 1</td>
<td>Bioc2001 Bot2004 Phil2080</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(Sum-Wts (Chemistry Physics Mathematics Statistics Computer-science) At-least 2))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(wt: 3)</td>
<td></td>
</tr>
<tr>
<td>Year 2 Session 2</td>
<td>Bioc2004 Bot2002 Phil3053</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(wt: 3)</td>
<td></td>
</tr>
<tr>
<td>Year 3 Session 1</td>
<td>Bioc3005 Bioc3007 Bot3009</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(wt: 3)</td>
<td></td>
</tr>
<tr>
<td>Year 3 Session 2</td>
<td>Bioc3002 Bioc3008 Bot3005</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(wt: 3)</td>
<td></td>
</tr>
</tbody>
</table>

**S9 AC:** 
\{(Sum-Wts (Chemistry Physics Mathematics Statistics Computer-science) At-least 2))\}

**C:** (Sum-Wts Yr1S1 4)

**EX:** \{Comp1002\}

**X:** \{(T Comp1002) (Comp1002 Comp1001)\}


**PC:** \{(Sum-Wts Yr1S2 4) (Sum-Wts Yr2S1 3) (Sum-Wts Yr2S2 3) (Sum-Wts Yr3S1 3) (Sum-Wts Yr3S2 3) (Sum-Wts A At-least 6) (Sum-Wts B At-least 2) (Sum-Wts C At-least 4) (Sum-Wts Science C At-least 2)\}
### Current Plan

| Year 1 Session 1 | Compl001 (wt: 4) |
| Year 1 Session 2 | Compl002 (wt: 4) |
| Over Year 1      | Chem001 Chem002 Bot001 |
| Year 2 Session 1 | Bioc2001 Bot2004 Phil2080 (Sum-Wts (Chemistry Physics Mathematics Statistics Computer-science) At-least 2)) (wt: 3) |
| Year 2 Session 2 | Bioc2004 Bot2002 Phil3053 (wt: 3) |
| Year 3 Session 1 | Bioc3005 Bioc3007 Bot3009 (wt: 3) |
| Year 3 Session 2 | Bioc3002 Bioc3008 Bot3005 (wt: 3) |

\[ S10 \ AC: \{(\text{Sum-Wts (Chemistry Physics Mathematics Statistics Computer-science) At-least 2))}\]  
C: Nil  
EX: {}  
X: {}  
CS: \{(T Comp002) (Comp002 Comp001) (T Phil3053) (Phil3053 Phil2080)  
(T Bot3009) (Bot3009 Bot2002) (Bot2002 Bot001)  
(Bot2002 (Sum-Wts (Chemistry Physics Mathematics Statistics Computer-science) At-least 2))  
(T Bot3005) (Bot3005 Bot2004) (Bot2004 Bot001)  
(T Chem1002) (T Bioc3007) (T Bioc3008)  
(Bioc3007 Bioc2004) (Bioc2004 Bioc2001)  
(Bioc2001 Chem1001) (Bioc3008 Bioc2004)  
(T Bioc3002) (T Bioc3005) (Bioc3002 Bioc2001)\}  
PC: \{(\text{Sum-Wts Yr1S1 4}) (\text{Sum-Wts Yr1S2 4}) (\text{Sum-Wts Yr2S1 3})  
(\text{Sum-Wts Yr2S2 3}) (\text{Sum-Wts Yr3S1 3}) (\text{Sum-Wts Yr3S2 3})  
(\text{Sum-Wts A At-least 6}) (\text{Sum-Wts B At-least 2})  
(\text{Sum-Wts C At-least 4}) (\text{Sum-Wts Science C At-least 2})\}
Appendix D. Illustration of Model-Search with Degree Planning

<table>
<thead>
<tr>
<th>Final Plan</th>
<th>Student Id: 9210042</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year 1 Session 1: Comp1001 (wt: 4)</td>
<td></td>
</tr>
<tr>
<td>Year 1 Session 2: Comp1002 (wt: 4)</td>
<td></td>
</tr>
<tr>
<td>Over Year 1: Chem1001 Chem1002 Bot1001</td>
<td></td>
</tr>
<tr>
<td>Year 2 Session 1: Bioc2001 Bot2004 Phil2080 (wt: 3)</td>
<td></td>
</tr>
<tr>
<td>Year 2 Session 2: Bioc2004 Bot2002 Phil3053 (wt: 3)</td>
<td></td>
</tr>
<tr>
<td>Year 3 Session 1: Bioc3005 Bioc3007 Bot3009 (wt: 3)</td>
<td></td>
</tr>
<tr>
<td>Year 3 Session 2: Bioc3002 Bioc3008 Bot3005 (wt: 3)</td>
<td></td>
</tr>
</tbody>
</table>

Planning Time: 108.31 secs on SPARC SLC workstation.

Note that the prerequisite constraint: (Sum-Wts (Chemistry Physics Mathematics Statistics Computer-science) At-least 2) which appears in the interval Yr2S1 is satisfied by units contained in the interval Yr2S1 and below it. Hence, the precedent restriction specified by the implicit constraints is satisfied. Also, note that there are many schedules possible for courses in each cumulative structure at the different states. First the schedules are found using course availability and course level heuristics. Failing this, brute force search is employed in finding a schedule. A successful search for one schedule for a set of courses shows that the set of courses is schedulable. The sequence of states $S_0, \ldots, S_{10}$ is the solution path.

At the completion of the above search the following menu is displayed for further constraints, and with the addition of some preferential constraints to the existing state a solution is searched anew.

Next Instruction?
1. Quit Constraint Addition/Retraction, Search for a Solution
2. Quit Planning
3. Add Constraints
4. Minimally Change Existing Solution
5. Display Formal Constraints
6. Display Preferential Constraints
7. Explain Solution

index> 3

Enter constraints in CL (? for help) > **exclude comp1003,**
**include comp3012,**
**not-take-obj-from philosophy**

Here are all the current preferential constraints...
1. (Exclude Comp1003)
2. (Include Comp3012)
3. (Not-take-obj-from Philosophy)

Next Instruction?
1. Quit Constraint Addition/Retraction, Search for a Solution
2. Quit Planning
3. Add Constraints
4. Minimally Change Existing Solution
5. Display Formal Constraints
6. Retract Constraints

index> 1

<table>
<thead>
<tr>
<th>Final Plan</th>
<th>Student Id: 9210042</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year 1 Session 1 :</td>
<td>Comp1001</td>
</tr>
<tr>
<td>(wt: 4)</td>
<td></td>
</tr>
<tr>
<td>Year 1 Session 2 :</td>
<td>Comp1002</td>
</tr>
<tr>
<td>(wt: 4)</td>
<td></td>
</tr>
<tr>
<td>Over Year 1 :</td>
<td>Chem1001 Chem1002 Bot1001</td>
</tr>
<tr>
<td>Year 2 Session 1 :</td>
<td>Comp2001 Bioc2001 Bot2004</td>
</tr>
<tr>
<td>(wt: 3)</td>
<td></td>
</tr>
<tr>
<td>(wt: 3)</td>
<td></td>
</tr>
<tr>
<td>Year 3 Session 1 :</td>
<td>Comp3012 Bioc3007 Bioc3005</td>
</tr>
<tr>
<td>(wt: 3)</td>
<td></td>
</tr>
<tr>
<td>Year 3 Session 2 :</td>
<td>Bioc3002 Bot3010 Bioc3008</td>
</tr>
<tr>
<td>(wt: 3)</td>
<td></td>
</tr>
</tbody>
</table>

Planning Time: 140.60 secs

This search process continues until the user is satisfied with a solution.
Appendix E

Illustration of IPLAN with Degree Planning

In this appendix, the use of IPLAN in the ANU BSc degree planning example is illustrated. A transcript of a session with IPLAN is given.

E.1 Constraints

The following formal constraints are assumed:

\[ FC = \left(\sum \text{Wts} \ A \ \text{At-least} \ 6\right) \left(\sum \text{Wts} \ (\text{Year1}) \ E \ \text{At-most} \ 2\right) \]
\[ \left(\sum \text{Wts} \ (\text{Year1}) \ (A \ E) \ \text{At-most} \ 10\right) \left(\sum \text{Wts} \ E \ \text{At-most} \ 6\right) \]
\[ \left(\sum \text{Wts} \ A \ \text{At-least} \ 6\right) \left(\sum \text{Wts} \ A \ P2 \ \text{At-most} \ 2\right) \]
\[ \left(\sum \text{Wts} \ Science \ \text{At-least} \ 12\right) \left(\sum \text{Wts} \ All \ \text{At-least} \ 20\right) \]
\[ \left(\sum \text{Wts} \ B \ \text{At-least} \ 2\right) \left(\sum \text{Wts} \ Science \ C \ \text{At-least} \ 2\right) \]
\[ \left(\sum \text{Wts} \ C \ \text{At-least} \ 4\right) \left(\sum \text{Wts} \ All \ \text{At-least} \ 20\right) \]
\[ \left(\sum \text{Wts} \ Yr3S2 \ \text{At-Most} \ 3\right) \left(\sum \text{Wts} \ Yr3S1 \ \text{At-Most} \ 3\right) \]
\[ \left(\sum \text{Wts} \ Yr2S2 \ \text{At-Most} \ 3\right) \left(\sum \text{Wts} \ Yr2S1 \ \text{At-Most} \ 3\right) \]
\[ \left(\sum \text{Wts} \ Yr1S2 \ \text{At-Most} \ 4\right) \left(\sum \text{Wts} \ Yr1S1 \ \text{At-Most} \ 4\right) \]
\[ \left(\sum \text{Wts} \ Yr3 \ \text{At-Most} \ 6\right) \left(\sum \text{Wts} \ Yr2 \ \text{At-Most} \ 6\right) \]
\[ \left(\sum \text{Wts} \ Yr1 \ \text{At-Most} \ 8\right) \]

E.2 IPLAN Problem Solving

Here, the interactive and incremental nature of the IPLAN approach is demonstrated in terms of degree planning. (Text in bold font denote user input.)

\[ >(iplan) \]
\[ /*\text{Invocation of IPLAN}*/ \]

Loading Knowledge Base......

....done!
Enter your student-id> 8910042
/*Any existing plan for the student is displayed here.*/

Instruction?
1. Automatic Search
2. Add Constraint
3. Query
4. Advise
5. Exit

index> 2

Enter constraint in CL (? for help)> not-take-obj-from Chemistry

Current Preferential constraints...
1. (Not-take-obj-from Chemistry)

Next Instruction?
1. Automatic Search
2. Add Constraint
3. Query
4. Advise
5. Exit
6. Retract Constraint

index> 2

Enter constraint in CL (? for help)> take-obj-from Mathematics,
        not-take-obj-from Botany,
        take-obj-from Computer-Science

Next Instruction?
1. Automatic Search
2. Add Constraint
3. Query
4. Advise
5. Exit
6. Retract Constraint

index> 2

Enter constraint in CL (? for help)> include comp3031,
        include comp1001,
        include Math1021
Current Preferential constraints...
1. (Not-take-obj-from Chemistry)
2. (Take-obj-from Mathematics)
3. (Take-obj-from Computer-Science)
4. (Include Comp3031)
5. (Include Comp1001)
6. (Include Math1021)
/*Order of selection of courses is important: the courses from earlier temporal intervals
are chosen before courses from later temporal intervals.*/

\[ a_0 = (\text{Include Comp1001}):\{(T \text{ Comp1001})\} \]

/*A unique label is created for each assumption.*/

\[ s_{0}: \text{label} : \{a_0\} \]
\[ \text{justification} : \{a_0\} \]
\[ \text{plan} : Yr1S1 \{\text{Comp1001}\} \]
\[ Yr1S2 \{\} \]
\[ Yr1 \{\} \]
\[ Yr2S1 \{\} \]
\[ Yr2S2 \{\} \]
\[ Yr2 \{\} \]
\[ Yr3S1 \{\} \]
\[ Yr3S2 \{\} \]
\[ Yr3 \{\} \]

\[ a_1 = (\text{Include Math1021}):\{(T \text{ Math1021})\} \]

\[ s_{1}: \text{label} : \{a_0, a_1\} \]
\[ \text{justification} : \{s_{0}, a_1\} \]
\[ \text{plan} : Yr1S1 \{\text{Comp1001, Math1021}\} \]
\[ Yr1S2 \{\} \]
\[ Yr1 \{\} \]
\[ Yr2S1 \{\} \]
\[ Yr2S2 \{\} \]
\[ Yr2 \{\} \]
\[ Yr3S1 \{\} \]
\[ Yr3S2 \{\} \]
\[ Yr3 \{\} \]
E.2. IPLAN Problem Solving

\[ a_2 = \text{(Include Comp3031):\{(T \text{ Comp3031}) (Comp3031 Comp2012)} \]
\[ \text{(Comp3031 (Sum-Wts (Mathematics Statistics) B At-least 1))} \]
\[ \text{(Comp2012 Comp2011) (Comp2011 Comp1002)} \]
\[ \text{(Comp2011 (Sum-Wts Mathematics At-least 2))} \]
\[ \text{(Comp1002 Comp1001)} \]

\text{sn2: label} : \{a_0, a_1, a_2\}
\text{justification} : \{sn0, a_2\}

\text{plan} : \begin{align*}
Yr1S1 & \{\text{Comp1001, Math1021}\} \\
Yr1S2 & \{\text{Comp1002, (Sum-Wts Mathematics At-least 2)}\} \\
Yr1 & \{\} \\
Yr2S1 & \{\text{Comp2011}\} \\
Yr2S2 & \{\text{Comp2012, (Sum-Wts (Mathematics Statistics) B At-least 1)}\} \\
Yr2 & \{\} \\
Yr3S1 & \{\text{Comp3031}\} \\
Yr3S2 & \{\} \\
Yr3 & \{\} \\
\end{align*}

/*Note (Sum-Wts (Mathematics Statistics) B At-least 1) is a prerequisite of Comp3031 and hence, it can appear in any interval before the interval in which Comp3031 appears.*/

Current Preferential constraints...
1. (Not-take-obj-from Chemistry)
2. (Take-obj-from Mathematics)
3. (Take-obj-from Computer-Science)
4. (Include Comp3031)
5. (Include Comp1001)
6. (Include Math1021)

Next Instruction?
1. Automatic Search
2. Add Constraint
3. Query
4. Advise
5. Exit
6. Retract Constraint
Appendix E. Illustration of IPLAN with Degree Planning

7. Explain
8. Minimally Change Existing Solution

index> 2

Enter constraint in CL (? for help) > include chem1001

Conflict: 1. (Include Chem1001)
2. (Not-take-obj-from Chemistry)

Retract one constraint from the above.

index> 1

/* Retraction of 1 leads to backtracking to sn2 and retraction of 2 leads to dependency-directed backtracking to the state which has first instance of a Chemistry course in its justification, the justification is revised, the working set of courses (WS) is updated and search is re-started from this state. */

sn2: label : {a0, a1, a2}
   justification : {sn1, a2}
   plan : Yr1S1 {Comp1001, Math1021}
         Yr1S2 {Comp1002}
         Yr1  {}
         Yr2S1 {Comp2011}
         Yr2S2 {Comp2012, (Sum-Wts (Mathematics Statistics) B At-least 1))}
         Yr2  {}
         Yr3S1 {Comp3031}
         Yr3S2 {}
         Yr3  {}

Current Preferential constraints...
1. (Not-take-obj-from Chemistry)
2. (Take-obj-from Mathematics)
3. (Take-obj-from Computer-Science)
4. (Include Comp3031)
5. (Include Comp1001)
6. (Include Math1021)

Next Instruction?
1. Automatic Search
2. Add Constraint
3. Query
4. Advise
5. Exit
6. Retract Constraint
7. Explain
8. Minimally Change Existing Solution

index> 2

Enter constraint in CL (? for help)> include comp2013, include Math2026

\[ a3 = (\text{Include Comp2013}):\{(T \text{ Comp2013}) \text{ (Comp2013 Comp1002)}\) \\
\( \text{ (Comp1002 Comp1001)}\}\]

/*Note in the choice of assumptions for satisfying the constraints, those assumptions are chosen which result in unique supports to courses in the existing solution and which maximally matches with the existing solutions.*/

sn3: label : \{a0, a1, a2, a3\}
justification : \{sn2, a3\}
plan : Yr1S1 \{Comp1001, Math1021\}
  Yr1S2 \{Comp1002, (Sum-Wts Mathematics At-least 2)\}
  Yr1  \{\}
Yr2S1 \{Comp2011, Math2023\}
Yr2S2 \{Comp2012, Comp2013, (Sum-Wts (Mathematics Statistics) B At-least 1))\}
  Yr2  \{\}
Yr3S1 \{Comp3031\}
Yr3S2 \{\}
Yr3  \{\}

\[ a4 = (\text{Include Math2026}):\{(T \text{ Math2026}) \text{ (Math2026 Phys1001)}\}\]

sn4: label : \{a0, a1, a2, a3, a4\}
justification : \{sn3, a4\}
plan : Yr1S1 \{Comp1001, Math1021\}
  Yr1S2 \{Comp1002, (Sum-Wts Mathematics At-least 2)\}
Appendix E. Illustration of IPLAN with Degree Planning

Yr1 \{Phys1001\}
Yr2s1 \{Comp2011, Math2023\}
Yr2s2 \{Comp2012, Comp2013, Math2026,
    (Sum-Wts (Mathematics Statistics) B At-least 1)\}\}
Yr2 \{\}
Yr3s1 \{Comp3031\}
Yr3s2 \{\}
Yr3 \{\}

Current Preferential constraints...
1. (Not-take-obj-from Chemistry)
2. (Take-obj-from Mathematics)
3. (Take-obj-from Computer-Science)
4. (Include Comp3031)
5. (Include Comp1001)
6. (Include Math1021)
7. (Include Comp2013)
8. (Include Math2026)

Next Instruction?
1. Automatic Search
2. Add Constraint
3. Query
4. Advise
5. Exit
6. Retract Constraint
7. Explain
8. Minimally Change Existing Solution

index> 2

Enter constraint in CL (? for help) > what-if include Comp3036

\[\text{anc1 = (Include Comp3036):\{(T Comp3036) (Comp3036 Comp2012)\}}\]
\[\text{(Comp3036 (Sum-Wts (Mathematics Statistics) B At-least 1))}\]
\[\text{(Comp2012 Comp2011) (Comp2011 Comp1002)}\]
\[\text{(Comp2011 (Sum-Wts Mathematics At-least 2)) (Comp1002 Comp1001)}\]
E.2. IPLAN Problem Solving

\text{sncl: label : \{a0, a1, a2, a3, a4, anc1\}}
\text{justification : \{sn4, anc1\}}
\text{plan : Yr1S1 \{Comp1001, Math1021\}}
\text{Yr1S2 \{Comp1002, (Sum-Wts Mathematics At-least 2)\}}
\text{Yr1 \{Phys1001\}}
\text{Yr2S1 \{Comp2011, Math2023\}}
\text{Yr2S2 \{Comp2012, Comp2013, Math2026, (Sum-Wts (Mathematics Statistics) B At-least 1)\}}
\text{Yr2 \{\}}
\text{Yr3S1 \{Comp3031, Comp3036\}}
\text{Yr3S2 \{\}}
\text{Yr3 \{\}}

Accept this state \(< y, n >\)? \text{y}
/*If the answer is \text{n} then the control is returned to the previous state and problem-solving resumes from there.*/

\[ a5 = (Include \text{Comp3036}):\{(T \text{Comp3036}) \text{Comp3036 Comp2012} \}
\text{(Comp3036 (Sum-Wts (Mathematics Statistics) B At-least 1))}
\text{(Comp2012 Comp2011) (Comp2011 Comp1002) (Comp2011 (Sum-Wts Mathematics At-least 2)) (Comp1002 Comp1001))} \]

\text{sn5: label : \{a0, a1, a2, a3, a4, a5\}}
\text{justification : \{sn4, a5\}}
\text{plan : Yr1S1 \{Comp1001, Math1021\}}
\text{Yr1S2 \{Comp1002, (Sum-Wts Mathematics At-least 2)\}}
\text{Yr1 \{Phys1001\}}
\text{Yr2S1 \{Comp2011, Math2023\}}
\text{Yr2S2 \{Comp2012, Comp2013, Math2026, (Sum-Wts (Mathematics Statistics) B At-least 1)\}}
\text{Yr2 \{\}}
\text{Yr3S1 \{Comp3031, Comp3036\}}
\text{Yr3S2 \{\}}
\text{Yr3 \{\}}

Current Preferential constraints...
1. (Not-take-obj-from Chemistry)
Appendix E. Illustration of IPLAN with Degree Planning

2. (Take-obj-from Mathematics)
3. (Take-obj-from Computer-Science)
4. (Include Comp3031)
5. (Include Comp1001)
6. (Include Math1021)
7. (Include Comp2013)
8. (Include Math2026)
9. (include Comp3036)

Next Instruction?
1. Automatic Search
2. Add Constraint
3. Query
4. Advise
5. Exit
6. Retract Constraint
7. Explain

index > 6

Current Preferential constraints...
1. (Not-take-obj-from Chemistry)
2. (Take-obj-from Mathematics)
3. (Take-obj-from Computer-Science)
4. (Include Comp3031)
5. (Include Comp1001)
6. (Include Math1021)
7. (Include Comp2013)
8. (Include Math2026)
9. (include Comp3036)

Enter the index of constraint to retract: index > 2

Current Preferential constraints...
1. (Not-take-obj-from Chemistry)
2. (Take-obj-from Computer-Science)
3. (Include Comp3031)
4. (Include Comp1001)
5. (Include Math1021)
6. (Include Comp2013)
7. (Include Math2026)
8. (include Comp3036)

/*Note retraction of a constraint does not change the consistency of the existing solution.*/

Next Instruction?
1. Automatic Search
2. Add Constraint
3. Query
4. Advise
5. Exit
6. Retract Constraint
7. Explain

index> 3

1. ShowConstraints
2. ShowPlanObjects
3. Exit

index> 2

Enter plan object id or a set of attributes (? for help)>
(computer-science Mathematics), year3

1. ShowConstraints
2. ShowPlanObjects
3. Exit

index> 2

Enter plan object id or a set of attributes (? for help)>
comp3017
Appendix E. Illustration of IPLAN with Degree Planning

<table>
<thead>
<tr>
<th>Course Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit-code: Comp3017</td>
</tr>
<tr>
<td>Unit-name: Computer Science C17</td>
</tr>
<tr>
<td>Unit-title: Operating Systems</td>
</tr>
<tr>
<td>Group-number: C</td>
</tr>
<tr>
<td>Department: Computer-Science</td>
</tr>
<tr>
<td>Level: Year3</td>
</tr>
<tr>
<td>Points-value: 1</td>
</tr>
<tr>
<td>Session-offered: 1/2</td>
</tr>
<tr>
<td>Duration: 1 session(s)</td>
</tr>
<tr>
<td>Prerequisites: (And (Comp2012) (Sum-Wts (Mathematics Statistics) B At-least 1))</td>
</tr>
<tr>
<td>Incompatible-units: {}</td>
</tr>
</tbody>
</table>

1. ShowConstraints
2. ShowPlanObjects
3. Exit

index> 3

Current Preferential constraints...
1. (Not-take-obj-from Chemistry)
2. (Take-obj-from Computer-Science)
3. (Include Comp3031)
4. (Include Comp1001)
5. (Include Math1021)
6. (Include Comp2013)
7. (Include Math2026)
8. (include Comp3036)

Next Instruction?
1. Automatic Search
2. Add Constraint
3. Query
4. Advise
5. Exit
6. Retract Constraint
7. Explain
Current Preferential constraints...
1. (Not-take-obj-from Chemistry)
2. (Take-obj-from Computer-Science)
3. (Include Comp3031)
4. (Include Comp1001)
5. (Include Math1021)
6. (Include Comp2013)
7. (Include Math2026)
8. (include Comp3036)

Next Instruction?
1. Automatic Search
2. Add Constraint
3. Query
4. Advise
5. Exit
6. Retract Constraint
7. Explain

Enter constraint in CL (? for help) > **include comp3017**

\[ a6 = (\text{Include Comp3017});\{(T \text{ Comp3017}) (\text{Comp3017 Comp2012}) \]
\[ (\text{Comp3017 (Sum-Wts (Mathematics Statistics) B At-least 1)}) \]
\[ (\text{Comp2012 Comp2011}) (\text{Comp2011 Comp1002}) (\text{Comp1002 Comp1001}) \]
\[ (\text{Comp2011 (Sum-Wts (Mathematics Statistics) At-least 2)}) \}\]

\[ \text{sn6: label : \{a0, a1, a2, a3, a4, a5, a6\}} \]
\[ \text{justification : \{sn5, a6\}} \]
\[ \text{plan : Yr1S1 \{Comp1001, Math1021\}} \]
\[ \text{Yr1S2 \{Comp1002, (Sum-Wts Mathematics At-least 2)\}} \]
\[ \text{Yr1 \{Phys1001\}} \]
\[ \text{Yr2S1 \{Comp2011, Math2023\}} \]
\[ \text{Yr2S2 \{Comp2012, Comp2013, Math2026, (Sum-Wts} \]
\[ (\text{Mathematics Statistics) B At-least 1)}) \]
\[ \text{Yr2 \{\}} \]
Appendix E. Illustration of IPLAN with Degree Planning

Yr3S1 {Comp3031, Comp3036}
Yr3S2 {Comp3017}
Yr3  {}

Current Preferential constraints...
1. (Not-take-obj-from Chemistry)
2. (Take-obj-from Computer-Science)
3. (Include Comp3031)
4. (Include Comp1001)
5. (Include Math1021)
6. (Include Comp2013)
7. (Include Math2026)
8. (Include Comp3036)
9. (Include Comp3017)
Next Instruction?
1. Automatic Search
2. Add Constraint
3. Query
4. Advise
5. Exit
6. Retract Constraint
7. Explain

index> 2

Enter constraint in CL (? for help)> include Math1023,
include phys1001

a7 = (Include Math1023):{(T Math1023)}

sn7: label : {a0, a1, a2, a3, a4, a5, a6, a7}
justification : {sn6, a7}
plan : Yr1S1 {Comp1001, Math1021, Math1023}
  Yr1S2 {Comp1002, (Sum-Wts Mathematics At-least 2)}
  Yr1  {Phys1001}
Yr2S1 {Comp2011, Math2023}
Yr2S2 {Comp2012, Comp2013, Math2026, (Sum-Wts
(Mathematics Statistics) B At-least 1)}}
Yr2  {}
Yr3S1 {Comp3031, Comp3036}
Yr3S2 {Comp3017}
Yr3  {}

\[
a8 = (\text{Include Bot1001}):\{(T \text{ Bot1001})\}
\]

sn8: label : \{a0, a1, a2, a3, a4, a5, a6, a7, a8\}
justification : \{sn7, a8\}
plan : Yr1S1 \{Comp1001, Math1021, Math1023\}
Yr1S2 \{Comp1002, (Sum-Wts Mathematics At-least 2)\}
Yr1  \{Phys1001, Bot1001\}
Yr2S1 \{Comp2011, Math2023\}
Yr2S2 \{Comp2012, Comp2013, Math2026, (Sum-Wts
(Mathematics Statistics) B At-least 1)\}
Yr2  {}
Yr3S1 {Comp3031, Comp3036}
Yr3S2 {Comp3017}
Yr3  {}

**Current Preferential constraints...**
1. (Not-take-obj-from Chemistry)
2. (Take-obj-from Computer-Science)
3. (Include Comp3031)
4. (Include Comp1001)
5. (Include Math1021)
6. (Include Comp2013)
7. (Include Math2026)
8. (Include Comp3036)
9. (Include Comp3017)
10. (Include Math1023)
11. (Include Bot1001)

Next Instruction?
1. Automatic Search
2. Add Constraint
3. Query
4. Advise
5. Exit
6. Retract Constraint
7. Explain

index> 2

Enter constraint in CL (? for help)> include bot1001

\[
a_9 = \text{(Include Bot1001)} : \{(T \text{ Bot1001})\}
\]

\[
\text{sn9: label} : \{a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9\}
\]

\[
\text{justification} : \{[\text{Bsn8, a9}]\}
\]

plan : fail

nogood : \{\{(\text{Sum-Wts Yr1S1 At-most 4})\}, \{\text{Comp1001, Math1021, Math1023, Comp1002, Math1012, Phys1001, Bot1001}\}\}

/*Backtrack to previous state and resume search from there.*/

\[
a_8 = \text{(Include Bot1001)} : \{(T \text{ Bot1001})\}
\]

\[
\text{sn8: label} : \{a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8\}
\]

\[
\text{justification} : \{\text{sn7, a8}\}
\]

plan : Yr1S1 \{\text{Comp1001, Math1021, Math1023}\}

\[
\text{Yr1S2} \{\text{Comp1002, (Sum-Wts Mathematics At-least 2)}\}
\]

\[
\text{Yr1} \{\text{Phys1001, Bot1001}\}
\]

\[
\text{Yr2S1} \{\text{Comp2011, Math2023}\}
\]

\[
\text{Yr2S2} \{\text{Comp2012, Comp2013, Math2026, (Sum-Wts (Mathematics Statistics) B At-least 1)}\}
\]

\[
\text{Yr2} \{}\}
\]

\[
\text{Yr3S1} \{\text{Comp3031, Comp3036}\}
\]

\[
\text{Yr3S2} \{\text{Comp3017}\}
\]

\[
\text{Yr3} \{}\}
\]

Current Preferential constraints...
1. (Not-take-obj-from Chemistry)
2. (Take-obj-from Computer-Science)
3. (Include Comp3031)
4. (Include Comp1001)
5. (Include Math1021)
Extension of the existing plan with the following set of plan objects, as derived from COBWEB clusters, satisfy all the constraints:


A plan from the extension is:

\[
\begin{align*}
\text{plan : Yr1} & \{\text{Comp1001, Math1021, Math1023}\} \\
\text{Yr1} & \{\text{Phys1001}\} \\
\text{Yr2} & \{\text{Comp2011, Math2021, Math2023}\} \\
\text{Yr2} & \{\text{Comp2012, Comp2013, Math2026, Math2024}\} \\
\text{Yr3} & \{\text{Comp3036, Math3023, Math3027, Math3025, Math3021, Comp3036, Comp3031, Comp3015}\} \\
\text{Yr3} & \{\text{Comp3017, Math3032, Math3022, Comp3034, Comp3032}\}
\end{align*}
\]

1. Accept-Advice
2. Generate-Another-Advice
3. Disregard-Advice-Continue

/*If there is no advice then the interactive problem solving continues. If the advice is not accepted then problem solving continues in the interactive, exploratory manner and when all the preferential constraints are exhausted and a solution not found then automatic search through model-search1 is invoked. model-search1 performs an exhaustive search for a solution in an informed manner.*/
## Final Plan

<table>
<thead>
<tr>
<th>Year 1 Session 1</th>
<th>Comp1001 Math1023 Math1021 (wt: 4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year 1 Session 2</td>
<td>Comp1002 Math1024 Math1022 (wt: 4)</td>
</tr>
<tr>
<td>Over Year 1</td>
<td>Phys1001</td>
</tr>
<tr>
<td>Year 2 Session 1</td>
<td>Math2023 Math2021 Comp2011 (wt: 3)</td>
</tr>
<tr>
<td>Year 2 Session 2</td>
<td>Math2026 Math2024 Comp2013 Comp2012 (wt: 3)</td>
</tr>
<tr>
<td>Year 3 Session 1</td>
<td>Math3023 Math3027 Math3025 Math3021 Comp3036 Comp3031 (wt: 3)</td>
</tr>
<tr>
<td>Year 3 Session 2</td>
<td>Comp3017 Math3032 Math3022 Comp3034 Comp3032 (wt: 3)</td>
</tr>
</tbody>
</table>

Next Instruction?
1. Add Constraint
2. Query
3. Save Plan and Exit
4. Explain

/*The cumulative structure in the label and the schedule are saved for the student.*/

To illustrate dependency-directed backtracking, if a new constraint (Not-take-obj-from Physics) is added to the above final solution then the plan will become inconsistent with the constraint because it contains a Physics unit. Automatic resolution of this constraint involves dependency-directed backtracking to the state node sn4, revision of a4 such that it does not contain Phys1001—the new a4 becomes (Include Math2026):{(T Math2026) (Math2026 Math1012)}—and resuming problem solving from sn4 with the new justification.
Appendix F

Miscellaneous Algorithms and Functions

In this appendix, miscellaneous functions referenced in many algorithms in the thesis are described. Also, the scheduling algorithms of Chapter 4 are given.

F.1 Miscellaneous Functions

- add takes an object \( O \) and a sequence \( Seq \) and returns a sequence with \( O \) added to the front of \( Seq \).

- push takes an object \( O \) and a sequence \( Seq \) and returns another sequence with \( O \) added to the front of \( Seq \).

- pushb takes an object \( O \) and a sequence \( Seq \) and returns another sequence with \( O \) added to the back of \( Seq \).

- pop takes a sequence \( Seq \) and returns the front element of \( Seq \). The front element of \( Seq \) is removed as a side-effect.

- popb takes a sequence \( Seq \) and returns the last element of \( Seq \). The last element of \( Seq \) is removed as a side-effect.

- choose takes a set and returns one element from it.

- intersect takes two sets of objects and returns the set of elements common to both the sets.

- get-set takes a sequence of objects and returns a set of the objects.

- filter-obj takes two sets of objects, \( ST1 \) and \( ST2 \), and returns the set difference, \( ST1 \setminus ST2 \)
• *count-objs* finds the cardinality of a set of plan objects.

• *get-plan-objs* takes a structure and returns a set of plan objects which are present in the structure.

\[
\text{get-plan-objs: Structures} \rightarrow P \text{ Plan-Objs,}
\]
where *Structures* represents a set of *SStruct*, *II*, *CSS*, *A*, *λt*, *schedule* and *P* structures.

• *get-constraints* takes a structure and returns a set of constraints which are present in the structure.

\[
\text{get-constraints : Structures} \rightarrow P \text{ CONST.}
\]

• *get-schedule-cons* takes a structure and returns all the scheduling constraints in it.

• *filter-incomp-structs* takes a set of structures and a set of incompatible constraints, filters any structures comprising of incompatible plan objects and returns a set of *incompatibly-consistent* structures.

\[
\text{filter-incomp-structs: P CONST} \times P \text{ Structures} \rightarrow P \text{ Structures.}
\]

• *combine-scheds* takes two schedules, combines them and returns the combined schedule.

• The functions: *get-incomp-cons*, *get-pref-cons*, and *get-temp-cons* take a state as their argument and return a set of incompatible constraints, preferential constraints, and temporal constraints respectively.

### F.2 Scheduling Algorithms
extend-schedule: CSS × Schedules × P CONST → Schedules U {fail}
Input: css:CSS, constraints ⊆ CONST, sched ∈ Schedules

cons ← (get-schedule-cons constraints)
(*Collect all Sched-Cons from the existing set of constraints.*)

cons1 ← (get-schedule-cons css) U cons
(*Update the set of scheduling constraints with any that emerge from css.*)

new-sched ← (add-struct-to-sched css sched cons1)
(*Add css to the existing schedule.*)

if new-sched ≠ nil then
(return new-sched)
else
(add-to-nogoods cons1 (get-plan-objs sched) U (get-plan-objs css))
(*Add the set of all the scheduling constraint labels and the set of plan objects in the extended structure to the nogoods database.*)
(return fail)
(*The extended structure is not schedulable for the existing set of scheduling constraints.*)
fi

Table F.1: Extend-Schedule Algorithm.

add-struct-to-sched: CSS × Schedules × P CONST → Schedules U {fail}
Input: css:CSS, sched ∈ Schedules, cons1 ⊆ CONST, interval-cons is a set of unused temporal intervals (interval-cons ⊆ TI)

cand ← (get-supp-struct css)

nsched ← (get-schedule (get-cand-struct cand (get-avail-prec-coin-sched-pos-cons cons)) interval-cons (get-plan-time-cons cons))
(*get-avail-prec-coin-sched-pos-cons gets all the user-specified precedence, coincidence and schedule-position constraints from the existing set of constraints. get-plan-time-cons gets the plan-time constraint from the existing set of constraints.*)

new-sched ← (combine-scheds sched nsched)

if ∀c ∈ cons1, (satisfies? new-sched c) then (return new-sched)
else (return fail)
fi

Table F.2: An Algorithm for Addition of a CSS to a Schedule.
schedule: $CSS \times P \text{CONST} \times P \text{TI} \rightarrow \text{Schedules} \cup \{\text{fail}\}$,
Input: $css:CSS$, $cons \subseteq \text{CONST}$, interval-$cons$ is a set of unused temporal intervals ($\text{interval}-cons \subseteq \text{TI}$)

\begin{align*}
\text{cand} & \leftarrow (\text{get-supp-struct} \ css) \\
\text{cand1} & \leftarrow (\text{remove-objs} \ (\text{get-objs} \ xplan) \ \text{cand}) \\
\text{cons1} & \leftarrow (\text{get-schedule-cons} \ \text{cand}) \cup \text{cons} \\
\text{sched} & \leftarrow (\text{get-schedule} \ (\text{get-cand-struct} \ \text{cand1} \ (\text{get-avail-prec-coin-sched-pos-cons} \ \text{cons})) \ \text{interval-cons} \ (\text{get-plan-time-cons} \ \text{cons})) \\
\text{if} \ \text{sched} = \text{fail} \ \text{then} & \\
& (\text{add-to-nogoods} \ \text{cons1} \ (\text{get-plan-objs} \ \text{css})) \\
& \text{return} \ \text{fail} \\
\text{fi}
\end{align*}

(*Nogoods database is globally maintained. It records minimal sets of plan objects, each of which is known to fail. Nogoods database is explained in detail in Chapter 5.*)

Table F.3: An Algorithm for Drawing a Schedule from a $CSS$. 
Appendix G

Publications

Below is the current list of publications by the author which report on initial work on the research described in this thesis.

D. P. Sharma. Formalisation and representation of university rules for an intelligent degree advisor system. *Australian Computer Science Communications*, 13(1), February 1991, University of New South Wales.


D. P. Sharma. DPLANX: A constraint-based planning system. *Australian Computer Science Communications*, 14(1), February 1992, University of Tasmania.


D. P. Sharma. ADEAN: A constraint-based advisory system. Accepted for publication as a poster paper in *Proceedings of International Conference on Tools for Artificial Intelligence ICTAI-91*, November 1991, San Jose, California.