CONSTRAINT-BASED TEMPORAL REASONING ALGORITHMS WITH APPLICATIONS TO PLANNING

By

Ioannis Tsamardinos

B.S., Computer Science Department, University of Crete, Heraclion, Crete, Greece, 1995

M.Sc., Intelligent Systems Program, University of Pittsburgh, 1998

Submitted to the Graduate Faculty of

School of Arts and Sciences, Intelligent Systems Program, in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2001

University of Pittsburgh

Faculty of Arts and Sciences

This dissertation was presented
by
Ioannis Tsamardinos

_____

It was defended on July 19th, 2001

_____

and was approved by

Professor Bruce G. Buchanan

_____

Professor Panos Chrysanthis

_____

Professor Gregory Cooper

_____

Professor Kirk Pruhs

_____

_____

Professor Martha E. Pollack (Committee Chairperson)

CONSTRAINT-BASED TEMPORAL REASONING ALGORITHMS WITH APPLICATIONS TO PLANNING

Ioannis Tsamardinos, PhD
University of Pittsburgh, July 2001

Abstract

A number of systems and applications model time explicitly and so require expressive temporal formalisms and efficient reasoning algorithms. In particular, an increasing number of planning systems deal with metric, quantitative time. This dissertation presents a set of new, efficient, provably correct algorithms for expressive temporal reasoning, based on the use of constraint satisfaction problems and techniques, and demonstrates how they can be used to improve the expressiveness and efficiency of planning and execution systems.

First, the Disjunctive Temporal Problem (DTP) is examined. The DTP is a very powerful and expressive constraint-based, quantitative temporal reasoning formalism that subsumes a number of other previous formalisms, and is capable of representing many planning and scheduling problems. A new algorithm for solving DTPs is presented, and fully implemented in a system called Epilitis. This algorithm achieves significant performance improvement (two orders of magnitude on benchmark problem sets containing randomly generated problems) relative to previous state-of-the-art solvers. In addition, DTP solving is thoroughly examined: alternative solving techniques are investigated, classified, and theoretically analyzed.

In addition to solving DTPs, the execution and dispatching of plans encoded as DTPs is addressed. A novel dispatch algorithm for such plans is presented; amongst other desirable properties, it retains all the available flexibility in the occurrence of events during execution. This is the first known dispatch algorithm for DTPs.

Next, new constraint-based temporal formalisms are defined that allow, for the first time, the representation of temporal and conditional plans, dealing simultaneously with the uncertainty of the outcome of observations and with expressive temporal constraints. The formalisms are named the Conditional Simple Temporal Problem (CSTP) and the Conditional Disjunctive Temporal Problem (CDTP). Appropriate consistency checking algorithms accompany the definitions. In addition, complexity results, theoretical analysis, and tractable subclasses are identified. We also note that both CDTP and the CSTP consistency checking can be translated to DTP consistency checking, indicating a strong theoretical connection among the various formalisms.

We subsequently employ the above results in temporal reasoning to address a number of previously unsolvable problems in planning in a conceptually clear, and potentially very efficient, way. Leveraging the expressivity of our new formalisms, we present planning algorithms that can deal with richly expressive plans that include temporal constraints and conditional execution contexts. In particular we solve the problems of plan merging, plan cost optimization, and in-context plan cost evaluation.

Dedication

According to Aristotle, your father gives you the "ζειν" (living) but it is your teacher who provides you with the "ευ ζειν" (living well). This dissertation is dedicated to my father, my first and most influential teacher, for not only giving me both "ζειν" and "ευ ζειν"; for sowing the seeds of ορθού λόγου ("right", scientific thought) in me, for encouraging my mind to relentlessly inquire and doubt, and for protecting my intellectual freedom by letting me form my own beliefs. It is also dedicated to my mother for all the countless sacrifices she made to raise me and finally to my sister to whom I owe my existence; but that's another story.

Αφιέρωση

Σύμφωνα με τον Αριστοτέλη, ο πατέρας σου σου δίνει το «ζειν» αλλά είναι ο δάσκαλός σου που σου παρέχει το «ευ ζειν». Η παρούσα διατριβή είναι αφιερωμένη στον πατέρα μου, όχι μόνο για το «ζειν», αλλά ως πρώτος και σημαντικότερος δάσκαλός μου και για το «ευ ζείν», για την εμφύτευση μέσα μου του ορθού λόγου, για την παρότρυνση της συνεχής αναζήτησης και αμφισβήτησης και για την διαφύλαξη της πνευματικής μου ελευθερίας αφήνοντας με να διαμορφώσω τις δικές μου πεποιθήσεις. Είναι επίσης αφιερωμένο στην μητέρα μου για τις αμέτρητες θυσίες που έκανε για να με αναθρέψει και την αδερφή μου στην οποία χρωστάω την ύπαρξή μου. Αλλά αυτό είναι μια άλλη ιστορία.

Acknowledgements

There are simply no words to overstate the role of my advisor, Professor Martha E. Pollack in writing this dissertation. I would have never been able to complete this work without her. Right from the start she has been providing financial security, an intellectually stimulating environment, mentoring, and a shield against all administrative problems. But, most importantly, she was always able to guide me to the scientifically right direction and provide the long-term vision, while at the same time follow the low-level technical details and be the hardest adversary. Her door was always open to all students, something that I have taken advantage of way too often, always to be welcomed with a smile; even under the most stressful times. Above all I thank her for her friendship.

I would also like to thank Nicola Muscettola and Paul Morris at NASA Ames for our collaboration, for starting me on temporal reasoning, and for being the source of a number of ideas and problems to work on. Also, Thierry Vidal for his collaboration and help on the Conditional Simple Temporal Problem. I would like to thank the University of Pittsburgh and University of Michigan students that I have worked with through the years, in particular Sailesh Ramakrishnan for the many interesting discussions on plan merging and plan cost evaluation in context and his help with the Disjunctive Temporal Problem (DTP) experiments, Philip Ganchev for motivating work on DTP execution, and the rest of the plan management group in University of Michigan. Special thanks to Colleen McCarthy for her help with the DTP experiments but mostly for her friendship during the hard Ph.D. years.

I would also like to thank my dissertation committee for the effort, advice, and comments, especially Professor Gregory Cooper for his suggestions for improvements, Professor Bruce Buchanan for his help with the introduction, and Professor Panos Chrysanthis for the collaboration on applications of DTP solving; Amedeo Cesta and his students for the interesting discusions on DTP solving. I am grateful to Sonia Leach for the DTP experiments graphs done at last minute notice, with the psychological support during the hardest last months, and with helping me to edit the post-defense modifications.

Finally, I would like to thank the University of Pittsburgh for giving me the opportunity to be in the graduate program, the Intelligent Systems Program for the excellent course-work, the University of Michigan for hosting me during the last year of my studies, NASA for giving me the opportunity to work on real planning problems and providing me with valuable experience, and the agencies DARPA, AFOSR, NSF, and the Andrew Mellon Pre-doctoral Fellowship Program for their financial support.

TABLE OF CONTEXT

Acronyms and Terms

Term	Meaning	Pages
ACG	Armando, Castellini, and Giunchiglia DTP solver/approach	64
BDD	Binary Decision Diagrams	151
CBI	Constraint-Based Interval planners	11, 158
CDB	Conflict Directed Backjumping pruning method	46
CDTP	Conditional Disjunctive Temporal Problem	148
CSP	Constraint Satisfaction Problem	23
CSTP	Conditional Simple Temporal Problem	117
DCSP	Dynamic Constraint Satisfaction Problem	35
DF	Deadline Formula	106
DTP	Disjunctive Temporal Problem	37
ET	Execution Table	106
FC	Forward Checking	32, 40
FC-Off	Forward Checking switch-off	55
NG	No-good learning, No-good learning pruning method	28, 57
NR	No-good recording (learning)	28
OC	Oddi and Cesta DTP solver/approach	64
RS	Removal of Subsumed Variables pruning method	49
SAT	Satisfiability Problem	24
SB	Semantic Branching pruning method	52
SK	Stergiou and Koubarakis DTP solver/approach	63
STN	Simple Temporal Network	16
STP	Simple Temporal Problem	15
STPU	Simple Temporal Problem with Uncertainty	21
TCSP	Temporal Constraint Satisfaction Problem	19
TQA	Temporally Qualified Assertion	11
TSAT	Temporal Satisfiability solver; ACG solver	80

LIST OF TABLES

LIST OF FIGURES

Chapter 1

1. INTRODUCTION

1.1 Motivation

Many computer science systems have to reason about time in one way or another since the world is dynamic. Time plays a fundamental role in causality, prediction, and action. In particular, *applications that reason about action*, e.g. planners, need to represent explicitly or implicitly the trajectory in time of the system and the environment. Examples include a medical application that models the evolution and treatment of a disease and a scheduling system for the observations to be made by the Hubble Space Telescope. *This dissertation presents a set of new, efficient, provably correct algorithms for expressive temporal reasoning, based on the use of constraint satisfaction problems and techniques, and demonstrates how they can be used to improve the expressiveness and efficiency of planning, scheduling and execution systems.*

One approach to represent and reason about time is to associate predicates with temporal primitives such as intervals, denoting that the predicate is true during the associated intervals. Indeed, this is the approach taken by most logic-based systems: classical planners, for example, associate a predicate with a causal-link that is supposed to remain true during the time interval that corresponds to the causal-link; schedulers associate the use of a resource with a time interval[1].

A common reasoning task that uses such representations is to query and/or ensure the necessary or possible temporal relations between the temporal primitives. For example, a classical planner would like to ensure that no step that threatens the predicate of a causal-link is allowed to be executed during the time interval associated with the causal-link. A scheduler would like to ensure that the time interval associated with the use of a non-sharable resource, e.g. the use of a printer, does not overlap another such interval. A natural language understanding system might check the consistency of a mystery story by querying whether the time interval of a murderer being at the crime scene overlaps with the time of the murder. A medical diagnosis and treatment system would like to ensure that the time interval separating the dosages of two incompatible drugs is long enough to minimize drug interactions. A vast range of applications need to represent and reason about temporal knowledge.

Reasoning with the temporal relations can be done through a purely temporal algebraic model. The dominant approach in the literature is to *impose constraints among the temporal primitives and model the temporal queries as appropriate temporal extensions of constraint satisfaction problems (**CSP**). This is what we will call the constraint-based approach to temporal reasoning, the key concept upon which our work is based.* The primitives and the constraints amongst them form classes of temporal problems such as the Simple Temporal Problem (**STP**) and the Disjunctive Temporal Problem (**DTP**). Associated with such temporal problems are two important reasoning tasks: *consistency checking* and *dispatching*, the

[1] This approach corresponds to the use of a reified logic (see Section fill 2.2.1 for other logic-based approaches and a discussion).

latter being pertinent to temporal problems that represent plans. This dissertation addresses both tasks. Consistency checking determines whether there is a time assignment to the primitives that respects all the constraints. Dispatching refers to the task of efficiently, incrementally, flexibly, and dynamically building solutions (i.e. time assignments that respect the constraints) once consistency has been determined.

Consistency checking is important in part because a number of different possible queries can typically be cast as a consistency checking problem, for example whether two temporal intervals overlap. For temporal problems that represent plans, consistency of the problem implies that the plan can be executed (moreover, inconsistency means that the plan cannot be executed in a way that satisfies the constraints). If a temporal problem representing a plan is proved consistent, a dispatcher then dynamically determines when to start and end plan actions, i.e. it can assign times to the temporal variables of the plan.

Unfortunately, previous temporal reasoning systems are not adequate to fully cover the needs of a large number of applications and many problems remain unresolved. There are two contributing factors. First, certain state-of-the-art formalisms lack efficient consistency checking algorithms and have no available dispatching algorithms. Second, current formalisms cannot even adequately model certain problems because they lack expressive power.

This dissertation contributes to solving both of these problems by (i) presenting algorithms and techniques that *improve the efficiency* of reasoning on existing temporal problems, and (ii) by defining new classes of temporal problems that *extend the expressiveness* of available temporal frameworks. Specifically, we present techniques that significantly improve the efficiency of consistency checking in DTPs, implementing these techniques in a DTP solver named *Epilitis*. We also *present a dispatching algorithm* for DTPs.

We improve the expressiveness of constraint-based temporal problems by defining two new classes of problems, Conditional Simple Temporal Problems (**CSTP**) and Conditional Disjunctive Temporal Problems (**CDTP**). These can be used to represent flexible, quantitative temporal constraints in conditional execution contexts. For example, we can constrain the duration and execution time of a task, and specify conditions under which the task should be executed (e.g. print a document which will take 3 to 5 minutes sometime between 1 pm and 8 pm, but only if the printer is working correctly). With our formalisms, it is possible, for the first time, to simultaneously represent quantitative temporal constraints and conditional execution contexts. We also develop the appropriate consistency checking algorithms for CSTPs and CDTPs and identify efficiently solvable subclasses of these problems. Interestingly, the efficiently solvable subclasses align with traditional conditional planning problems.

As a further result, since temporal problems can represent plans, the increased expressivity of our formalisms has direct consequences for *planning*. We show how several planning problems can be transformed into our CSTP and CDTP formalism, which then have conceptually clear solutions. In addition, there is also indication of computational benefits under such a transformation. In particular, we show how the constraint-based temporal approach can be used for *flexible plan dispatching, plan merging, plan cost optimization, and plan cost evaluation in context* with richly expressive plans that include quantitative temporal constraints and conditional branches.

Even though it is hard to predict the ramifications of this work, we believe the results in this dissertation will help pave the way to a new generation of temporal planning systems. Currently,

only a handful of planning systems can represent quantitative temporal information. *However, real-world situated planners [Ghallab and Laruelle 1994; Muscettola, Nayak et al. 1998] indicate a clear need for quantitative time representations*. We hope that the increased efficiency and expressivity of the formalisms presented in this thesis can contribute toward that goal.

1.2 Problem Statement and Solution Approach

> **The problem addressed in this dissertation is to provide efficient temporal reasoning algorithms and expressive temporal reasoning formalisms to fill the needs of a number of applications, one of which is planning, that we use as a case study to show the applicability of our algorithms and formalisms.**

The approach taken is to use the constraint satisfaction paradigm both to cast the temporal reasoning problems and as an inspiration of ideas, techniques, and methods to adapt from, or extend, to increase the efficiency of the algorithms. All the temporal formalisms employed make use of temporal variables that represent distinct and instantaneous events, i.e. time-points. Constraints among these variables can then be defined.

As far as the planning problems are concerned, the approach taken is to create a temporal variable (time-point) to represent each event in a plan, typically the start and end of an action, and to define suitable constraints to transform a planning problem to a temporal one. Other approaches to temporal reasoning are purely logical-based ones (see Section 2.2.1 for a comparison of logical-based temporal reasoning methods). However, these approaches typically cannot handle quantitative temporal constraints at the logic level, unless a constraint-based level is also used as in our approach.

Other approaches to representing the temporal information in a plan are to use intervals instead of time-points as the temporal primitives or both intervals and time-points. There are no obvious advantages to either of these approaches versus the time-points approach and the community is still debating on which one is the best.

1.3 An Illustrative Example

This section presents a simple example to illustrate the foundational issues underlying this work and the potential applications of the theoretical and algorithmic results. Since the necessary background information has not yet been presented, the example will be informal.

Suppose that a spacecraft has a plan to follow a specific trajectory. This goal is achieved by three steps: a thrusting step to bring the spacecraft to a certain position, a turn step starting at a specific moment when the position has been reached, and a thrusting step after the turn. Following the constraint-based approach to temporal reasoning and defining the instantaneous temporal events to correspond to the events of starting and ending the three actions in the plan, we can encode temporal information such as the following: the duration of the first thrusting is between 1 and 2 hours, the turn must start at 12 a.m. tomorrow, the second thrust step must follow the turn by at least 3 hours, etc. Figure 1-1 shows schematically these events represented by the circles of Plan 1, where A_S and A_E denote starting and ending action A respectively. All solid arrows denote ordering constraints between the events; the bold arrows are constraints from the start of a step to the end of the step. The meaning of the dashed arrows is explained below.

Now suppose that the mission control requests that the spacecraft achieve a new goal, that of taking a picture of an asteroid. The asteroid will become visible between 10 p.m. today and 5 a.m. tomorrow. Taking the picture lasts at least 1 minute and of course must occur while the asteroid is visible. This plan is represented as Plan 2 in Figure 1-1.

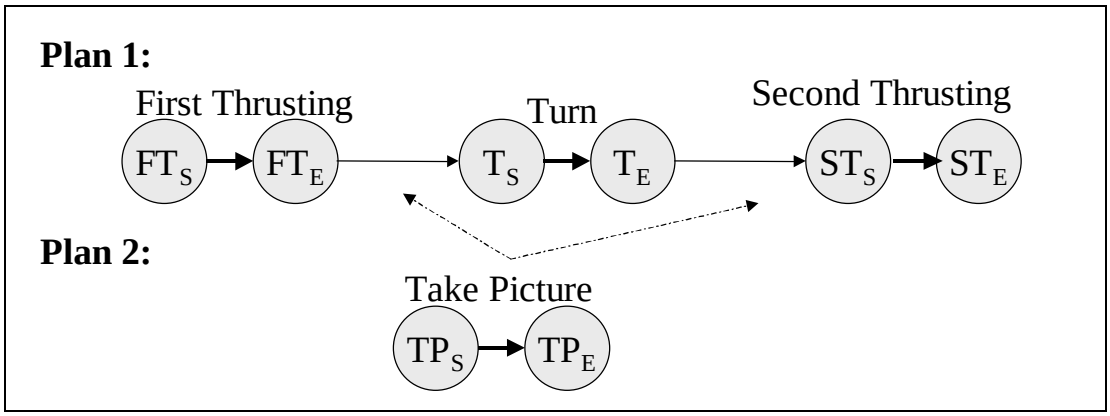


Figure 1-1: A simple planning example that illustrates the potential applications of our algorithms.

The problem of executing these two plans together is an instance of the plan merging problem that we address in Chapter 7. Our algorithms can analyze the domain knowledge and the plans being merged to infer the negative interactions among steps. In this example, thrusting and turning has the side effect of creating vibrations on the spacecraft and thus taking the picture should not overlap with any of the two thrusting steps or the turning step. Avoiding the negative interactions (also called resolving the conflicts) implies the additions of temporal constraints, in this case the constraint that interval $TP_S TP_E$ should not overlap with intervals $FT_S FT_E$, $T_S T_E$, or $ST_S ST_E$

The plan merging problem for non-temporal plans was previously solved, however we extend this work in Chapter 7 and *we show how to solve the plan merging problem for* ***temporal*** *plans including both qualitative and quantitative constraints* as in this example. In our approach, the merging problem is transformed to an instance of a Disjunctive Temporal Problem (DTP). Solving a DTP is equivalent to determining that there is a way to execute the corresponding plan, respecting all the constraints. *In Chapter 4 we thoroughly examine DTP solving and present a solver called Epilitis that is about two orders of magnitude faster than the previous state-of-the-art DTP solver on randomly generated DTPs.* We hope the efficiency improvements carry over to DTPs generated by real merging problems.

One solution in the above example is to take the picture between the first thrusting and before the turning step; another is to take it after the turn and before the second thrusting step, as the dashed arrows in the figure indicate. One could find an exact schedule, i.e. a specific time assignment to the events that respects the constraints, and execute the plan. Given the rigidity of an exact schedule in dynamic environments and the uncertainty of execution, it is desirable to adopt a more flexible approach. Existing algorithms are capable of commiting to one of the two general solutions we just presented (e.g. that the picture will be taken after the first thrusting step and before the turn) instead of adopting an exact schedule. *In Chapter 5, we present a least commitment algorithm that goes even further, retaining all solutions without committing to any particular one, guaranteeing maximally flexible execution.*

Slightly modifying the example above, suppose that we include an observation action that senses whether the camera works properly before attempting to take the picture. If so, the plan proceeds with taking the picture, otherwise a transmission step is executed that informs ground control about the problem. Reasoning with both these features (quantitative constraints and conditional executions) simultaneously was not possible using previous state of the art formalisms. *In Chapter 6 we present new formalisms, named CSTP and CDTP, that are able to represent and reason with this type of conditional execution and the quantitative temporal constraints simultaneously.* Our consistency checking algorithms in Chapter 6 can determine whether (and how) there is a way to execute the conditional temporal plan so that no matter what the outcome of the observations are, we are guaranteed to correctly complete execution.

In Chapter 7 we further use the CSTP and CDTP formalisms, not only to represent conditional temporal plans and determine their consistency, but also to be able to perform plan merging with this type of richly expressive plans. In addition, we show how to extend these algorithms to perform plan cost optimization and plan cost evaluation in context. For example, suppose that the spacecraft's robotic arm can be equipped with different tools. Two different plans might require the same tool, and so when merged, two preparation steps of changing the tool might be scheduled. Our plan cost optimization algorithm in Chapter 7 could determine if and how the two steps could be merged into one step so that the robotic arm can attach the tool only once and thus reduce power consumption.

Even this simple example illustrates the power and potential of our work. We used a planning example drawn from the space domain, but other areas that deal with time form prime candidate applications.

1.4 Overview and Contributions

The dissertation is organized as follows. Chapter 2 provides some basic planning and temporal reasoning background, including discussion of the Simple Temporal Problem (STP), the Temporal Constraint Satisfaction Problem (TCSP), and the Simple Temporal Problem with Uncertainty (STPU). Chapter 3 continues with background material on Constraint Satisfaction Problems (CSP), backtracking algorithms, and the no-good recording scheme that we will use in the later chapters. Chapter 8 concludes the dissertation with a discussion of the results and future directions of this work. The intermediate chapters with original material contain the following contributions:

Chapter 4: This chapter deals with consistency checking in Disjunctive Temporal Problems (**DTP**). *The main result of this chapter is an algorithm for checking the consistency of DTPs that is about two orders of magnitude faster than the previous state of the art algorithm on benchmark problems containing randomly generated DTPs. The algorithm has been fully implemented and tested in a system called Epilitis that provides flexibility to experiment with alternative solving strategies.*

- The DTP is defined and presented and basic solving techniques based on search are presented. Then, a complete literature survey follows, and all the pruning techniques previously used are analyzed and classified theoretically. We also analyze and theoretically compare all current DTP solvers and identify the trade-offs each one favors.

- We design new implementations that enable the simultaneous use of the different pruning techniques. This requires first identifying and resolving their negative interactions.
- We conduct experimental investigations of all known pruning techniques and compared their relative strength and the synergies among them. We theoretically compare the CSP versus the SAT approach in DTP solving.
- We introduce a new pruning technique to DTP solving, namely no-good recording (learning), by adopting it from the CSP literature, and we show that it significantly improves performance. We experiment with no-good recording and identify the optimal or a near optimal no-good recording size for the type of problems we use.
- We investigate various heuristics, some of which use information from the recorded no-goods, and identify the best heuristic.
- We provide experimental and theoretical evidence why FC-off, a technique that reduces the number of consistency checks in DTP solvers, does not necessarily increase performance.
- We identify potential problems with using forward checks (also called consistency checks) as a machine-independent measure of performance, which is currently the adopted measure in the literature. We counter suggest a more accurate measure that also takes into consideration the constraint propagations.
- We build a DTP solver that combines all the previous pruning methods together, adds no-good learning with optimally bounded no-good size, and employs the theoretically best way known so far of performing forward-checking. The resulting system, *Epilitis,* improves run-time performance about two orders of magnitude over the previous state-of-the-art DTP solver on randomly generated DTPs.

Chapter 5: This chapter deals with the problem of dispatching temporal plans represented as DTPs while retaining all the execution flexibility. *The main result of this chapter is a dispatching algorithm for DTPs with several desired properties.*

- The problem of DTP dispatching is identified and formalized and the desired properties for a dispatcher are defined, i.e. being ***correct, deadlock-free, maximally flexible,*** and ***useful***.
- A dispatching algorithm that provably solves the problem with the desired properties is presented.

Chapter 6: This chapter deals with the definition of new temporal reasoning frameworks that can reason with quantitative temporal constraints and conditional execution contexts. The new formalisms defined are called Conditional Simple Temporal Problem (**CSTP**) and Conditional Disjunctive Temporal Problem (**CDTP**). *The main result of this chapter is consistency-checking algorithms for the different notions of consistency in CSTP and CDTP. We prove their correctness and complexity and identify tractable problem subclasses.*

- We present formal definitions of CSTP and CDTP.
- We identify and define three notions of consistency, namely Strong Consistency, Weak Consistency, and Dynamic Consistency.
- We prove the result that Weak Consistency in CSTP and CDTP is co-NP-complete.
- We define a minimal execution scenario as a minimal set of observations that define which nodes should be executed and which should not, and provide algorithms for calculating the set of minimal execution scenarios without having to generate all the possible scenarios.
- We prove that checking Strong Consistency is equivalent to checking STP consistency.
- We provide an algorithm for checking Weak Consistency that, unlike the obvious brute force algorithm, performs incremental computation.
- We provide a general algorithm for determining Dynamic Consistency.
- We provide special cases of CSTPs where additional structural assumptions make the generation of the set of minimal execution scenarios computationally easy.
- For one special case of CSTPs, that we call CSTPs with typical conditional plan structure with no merges, we proved that Dynamic Consistency is equivalent to STP consistency of the CSTP.
- We determine other factors and structural assumptions that facilitate the determination of Dynamic Consistency.

Chapter 7: This chapter applies the constraint-based temporal reasoning paradigm to planning in order to solve several planning problems in a conceptually clear way and potentially increase efficiency. *The main result in this chapter are algorithms for the problems of plan merging, plan cost optimization, and plan cost evaluation in context for richly expressive plans with quantitative temporal constraints and conditional branches by casting them as DTPs or CDTPs problems.*

- We present a theory of conflicts and conflicts resolution in plans with quantitative temporal constraints and conditional branches..
- We develop an algorithm for plan merging for such richly expressive plans, making use of the theory of conflicts and casting their resolution as a DTP or CDTP (depending whether conditional branches are allowed or not in the plan).
- We develop an algorithm for plan cost optimization under certain clearly identified assumptions, which again converts the problem to a DTP or CDTP.
- We present an algorithm for plan cost evaluation in context again using a conversion to a DTP or CDTP.

The applicability of this work has already been proven. The roots and principles of our approach are found in the first autonomous spacecraft controller, NASA's Remote Agent, which flew on Deep Space I. All of the problems addressed are further inspired by and intended to be

applied to real planning projects we have been involved with, such as the Plan Management Agent, an intelligent calendar that manages the user's plans, and Nursebot, a robotic assistant intended to help elderly people with their daily activities. For some time, an early version of Epilitis has been a core component of the Plan Management Agent and Nursebot software and has proven extremely effective. The latest results presented here are currently being incorporated in both projects. We are also investigating the applicability of our techniques to areas such as temporal plan generation, Workflow Management Systems, Scheduling, and consistency checking of clinical protocols.

Chapter 2

2. BASIC PLANNING AND CONSTRAINT-BASED TEMPORAL REASONING BACKGROUND

This chapter reviews some of the basic planning and constraint-based temporal reasoning classes of problems for use in the subsequent chapters.

2.1 Related Work in Planning

Planning is one of the major and older fields of Artificial Intelligence and it has received much attention by AI researchers. The planning problem is defined as follows: given a description of available actions, of the initial state of the system, and of the desired goal, compute a description of a course of action that results in the goal state when executed in the initial state. This very broad and general definition of planning includes classical planning [Weld 1994], temporal planning [Ghallab and Laruelle 1994; Muscettola 1994; Penberthy and Weld 1994], probabilistic planning, conditional planning [Peot and Smith 1992], universal planning [Schoppers 1987], hierarchical task networks [Erol, Hendler et al. 1994], and cased-based reasoning planning [Hanks and Weld 1992]. This section is based on the excellent introductory and survey papers [Weld 1994; Weld 1999; Smith, Frank et al. 2000].

Most attention has been given to classical planning, in which the objective is to achieve a given set of goals, usually expressed as a set of positive and negative literals in the propositional calculus. The initial state of the world is also expressed as a set of literals. The possible actions are characterized using what is known as STRIPS operators, which are parameterized templates containing preconditions that must be true before the action can be executed and a set of changes or effects that the action will have on the world. Again, the preconditions and effects are positive or negative literals. A number of more recent planning systems have allowed an extended version of the STRIPS language known as ADL [Pednault 1989], which allows disjunctions in the preconditions, conditionals in the effects, and limited universal quantification in both the preconditions and effects.

Even though ADL extends the kind of problems that can be expressed in classical planning, all classical planners make a number of simplifying assumptions. These include assuming instantaneous, atomic actions (there is no explicit model of time), there is no provision for specifying the usage of resources, complete knowledge of the world and particularly of the initial state is assumed, along with a deterministic environment, while the goals are only goals of attainment and categorical. Additionally the only source of change in the world is the planning system. Attempts to extend the ADL representation and classical planners to include richer models

of time are in [Vere 1983; Penberthy and Weld 1994; Smith and Weld 1999], to allow various types of resources in [Penberthy and Weld 1994; Koehler 1998; Kautz and Walser 1999], to introduce forms of uncertainty into the representation in [Etzioni, Hanks et al. 1992; Peot and Smith 1992; Draper, Hanks et al. 1994; Kushmerick, Hanks et al. 1995; Golden and Weld 1996; Pryor and Collins 1996; Golden 1998; Smith and Weld 1998; Weld, Anderson et al. 1998], to introduce more interesting types of goals[Vere 1983; Williamson and Hanks 1994; Haddawy and Hanks 1998]. Also, a type of planning that incorporates the inclusion of probabilistic actions and the replacement of categorical goals with utility functions to be maximized is decision theoretical planning [Boutilier, Dean et al. 1995].

There are various ways to solve classical planning problems, or planning problems in general, most of which perform a search. One type of search is state-space and another is plan-space. State-space searches in the space of possible states the system might find itself in. It starts from the initial state and applies a STRIPS operator that leads the system to a different state, until we reach a state where all the goals are met; or it starts from any state where the goals are satisfied and considers which operators could have led the system to this state, searching backwards until the initial state is reached. Any path from the initial state to a goal state corresponds to a sequence of operators that is a valid plan. Another type of search is performed in the space of plans: starting with an empty plan we add operators until either a valid plan or a dead-end is reached; in the latter case, back-tracking then occurs.

Planners can also be divided into total order planners and partial order planners. Total order planners return sequences of actions totally ordered, while partial order plans return a partially ordered set of actions any serialization of which is a valid plan. Most commonly, state-space searching is associated with total-order planners while plan-space search is associated with partial order planners.

A ground-breaking advance in planning came with the Graphplan system [Blum and Furst 1997]. Graphplan employs a very different approach to searching for plans. The basic idea is to perform a kind of reachability analysis to rule out many of the combinations and sequences of actions that are not compatible. Starting with the initial conditions, Graphplan figures out the set of propositions and actions that are possible after one step, two steps, and so on, storing this information in a graph. The significant increase in performance that Graphplan exhibits arises from the fact that we can analyze the graph to infer sets of mutually exclusive propositions and actions. In essence the mutually exclusive conditions are constraints among propositions and actions. Thus, Graphplan strongly supports the idea of applying Constraint Satisfaction techniques in planning. Actually, the idea was first courted by [Joslin 1996] and subsequently proved viable and highly promising in [Yang 1990; Kautz and Selman 1992; Blum and Furst 1997; Kambhampati 2000]. It has since taken the community by storm [Weld 1999]. We also take a CSP approach to planning in this dissertation by considering the application of constraint-based temporal reasoning for plan-merging and plan generation at Chapter 7.

Another CSP-based approach to planning that is worth mentioning is planning as satisfiability [Kautz and Selman 1992; Kautz and Selman 1996], which translates the planning problem to a SAT problem with great performance gains.

For our purposes, of great interest are planners that can represent metric time information. While some planners directly extend the classical planning representation to encode action

duration for example [Vere 1983], other planners are based on an emerging paradigm called *Constraint-Based Interval* (CBI) approach. Rather than describing the world by specifying what facts are true in discrete time slices or states, these planners assert that propositions hold over intervals of time. Similarly, actions and events are described as taking place over intervals. The idea has its root in Allen's interval algebra and one of the first planners to follow the idea was Descartes by Joslin [Joslin 1996] who named these kind of intervals as *temporally qualified assertion* (TQA). The interval representation permits more flexibility and precision in specifying temporal relationships than is possible with simple STRIPS operators. For example, we can specify that a precondition only need hold for the first part of an action, or that some temporary condition will hold during the action itself.

The CBI approach shares many similarities to the partial order approach, but there are many differences too. For CBI planners, the temporal relationships and reasoning are often more complex. As a result CBI planners typically make use of an underlying constraint network to keep track of the TQAs and constraints in a plan. *For each interval two variables are introduced into the constraint network, corresponding to the beginning and ending points of the interval.* We will assume this representation in the rest of this dissertation, whenever we refer to plans where actions have temporal extent. Inference and consistency checking in such constraint networks can often be accomplished using STP representations and algorithms.

CBI planners can be viewed as dynamic constraint satisfaction engines since the planner alternately adds new TQAs and constraint to the network, then uses constraint satisfaction techniques to propagate the effects of those constraints to check for consistency. Planning systems that follow the CBI approach include Allen's Trains planner [Ferguson 1991; Ferguson, Allen et al. 1996], Joslin's Descartes [Joslin and Pollack 1995], the HSTS/Remote Agent planner [Muscettola 1994; Jonsson, Morris et al. 1999; Jonsson, Morris et al. 2000], and IxTeT [Ghallab and Laruelle 1994; Laborie and Ghallab 1995].

2.2 Related Work in Temporal Reasoning

Within AI, examples of fields that depend heavily on temporal reasoning include Medical Diagnosis and Expert Systems [Keravnov and Washbrook 1990; Perkins and Austing 1990; Rucker, Maron et al. 1990; Russ 1990; Aliferis and Cooper 1996], where a historical evolution of the symptoms of a disease is often necessary to determine the cause and appropriate treatment, Planning [Tsang 1987; Allen and Koomen 1990; Reichgelt and Shadbolt 1990; Allen 1991; Bacchus and Kabanza 1996; Bacchus and Kabanza 1996], where reasoning about actions and predicting their future effects is essential, and Natural Language Understanding [Grasso, Lesmo et al. 1990; Yip 1995], where verb tenses reveal time relationships that illuminate the meaning of sentences. This section provides a very brief overview of temporal reasoning, focusing on the issues and approaches most relevant to the rest of the dissertation. The overview is based on [Gennari 1998] and [Vila 1994].

[Vila 1994] defines Temporal Reasoning (TR) as formalizing the notion of time and providing means to represent and reason about the temporal aspects of knowledge. A TR framework should provide an *extension to the language* for representing the temporal aspects of the knowledge, as well as a *temporal reasoning system* to determine the truth of any temporal logical assertion. The language

extensions are typically logical based. We can conceptually split the responsibilities of the reasoning systems into two components: (i) finding the relationships between the time primitives, i.e. time points, time intervals, or both, and (ii) determining the truth-value of the predicates at each one of them. This dissertation deals primarily with the first task and the application of its results to planning.

2.2.1 Logic-based approaches to Temporal Reasoning

There are three main ways to introduce time in logic: first-order logic with temporal arguments, modal temporal logics, and reified temporal logics. The method of *temporal arguments* [Haugh 1987] consists of representing time just as another parameter in a first-order predicate calculus. Functions and predicates are extended with the additional temporal argument denoting the particular time at which they have to be interpreted. In philosophy the idea was first introduced by Russell [Russell 1903]; it was adapted for AI in the Situation Calculus [McCarthy and Hayes 1969] and the more recent work by Haugh [Haugh 1987]. The failure to give a special status to time, neither notational nor conceptual can be problematic as well as a limiting factor of its expressive power. For example, if the predicate *dance(x, y, t)* is used to denote *x* is dancing with *y* at time *t*, there is nothing to disallow *dance(Jordi, Iolanda, Maria)* as a legal formula (see [Vila 1994]).

In the *modal temporal logics* (MTL) the concept of time is included in the interpretation of the logic formula. The classical possible world semantics [Kripke 1963] is re-interpreted in the temporal context by making each possible world represent a different time. The language is an extension of the propositional or predicate calculus with modal temporal operators [Prior 1955] such as $F\varphi$ to denote φ is true in some future time, and $G\varphi$ to denote φ is true in every future time. Time elements are related by a temporal precedence relation and the standard modal operators of possibility and necessity are re-interpreted over future and past times. Although no assumption is made about the nature of time (points or intervals) the view of temporal individuals as points of time appears to be more natural (see [Halpern and Shoham 1986] for an exception). The notational efficiency of MTL makes it appealing for Natural Language Understanding applications where it has been widely used [Galton 1987]; it has also received very wide acceptance in programming theory (section 1.3 in [Galton 1987]). An advantage of MTL is that it can directly be combined with other modal qualifications that express belief, knowledge, etc. [Gaborit and Sayettat 1990], while a disadvantage is the current status of performance of MTL systems. See [Orgun and Ma 1994] for an overview of temporal and modal logic programming.

In *reified temporal logics* one can talk about the truth of temporally quantified assertions while staying in a first-order logic. Reifying logic involves translating into a meta-language, where a formula in the initial language, i.e. the object language, becomes a term – namely a propositional term – in the new language. Thus, in this new logic one can reason about the particular aspects of the truth of expressions of the object language through the use of truth predicates like TRUE. Typically, it is first-order logic that is the object language, although one could also use a modal logic. In reified logic, temporal formulas look like TRUE(*atemporal expression, temporal expression*), with the intended meaning that the first argument is true "during" the time denoted by the second argument ("during" can have a number of interpretations such as throughout the temporal

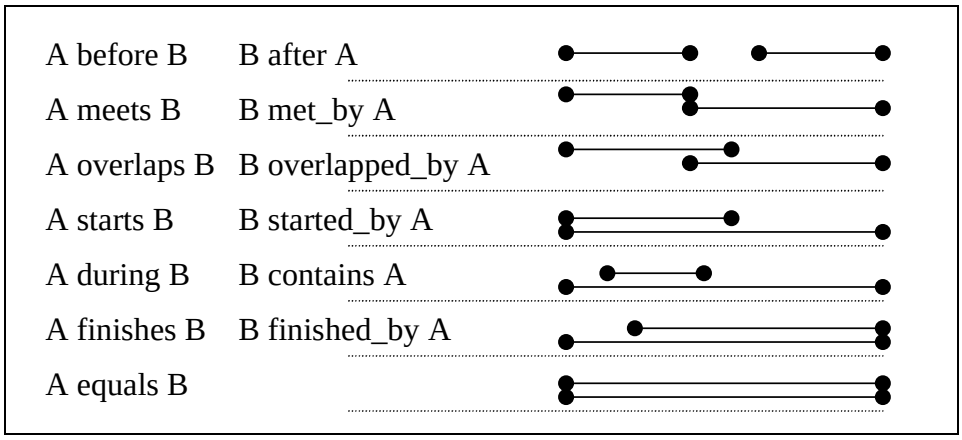


Figure 2-1: The 13 relations between intervals in Allen's algebra. Interval A is always either at the right or top of interval B.

expression, at some time during the temporal expression, etc.). For a more detail discussion on reified logics see [Vila 1994].

Any TR formalism establishes a link between atemporal assertions and a temporal reference. The temporal reference is made up of a set of temporal elements related by one or more temporal relations. Different primitives have been considered for the temporal elements with main candidates time points, time intervals, or a combination of both. After deciding on the ontological primitive, one may want to give structure to this ontology by providing axioms describing the behavior of the temporal relations that we will call the structure of time. Some choices that have appeared in the literature are discrete vs. dense time, bounded vs. unbounded, and linear, branching, partially ordered, parallel, and circular precedence relations.

2.2.2 Constrained-based Temporal Algebras

As mentioned in [Vidal 2000], in reified temporal logics the propositions are separated from their temporal qualifications. Queries such as "is proposition P true at time *t*?" are still processed at the logical level, but reasoning upon the temporal relations can be done through a purely temporal algebraic model. One usually gets a constraint network that is mainly used to check the temporal consistency of the given problem. In this thesis, we do not address the logic-based part of the temporal reasoning, but instead we tackle the algebraic, constraint-based, part of the reasoning.

We can separate the temporal algebras and the constraint systems that have been developed into two large classes: qualitative and metric (also called quantitative). One of the first qualitative algebras was Allen's algebra of intervals [Allen 1983]. It was based on the 13 mutually exclusive relations that can possibly exist between two intervals, as shown in Figure 2-1. In this algebra, time is linear and dense, and the time primitives are intervals. The constraints allowed between two time intervals A and B are of the form (A $\{r_1 \vee \ldots \vee r_n\}$ B) where r_i is any of the 13 primitive relations. Each such constraint defines the possible relation between A and B, e.g. (A {meets, after} B) means that A either meets or is after B. Allen provides a polynomial time algorithm for computing the closure of a set of constraints that is sound but not complete. Kautz et al. [Vilain and Kautz

1986] proves that checking consistency in the interval algebra is NP-Hard. They thus suggest using a *point algebra* in which consistency checking under certain assumptions is polynomial.

The point algebra is defined the same way the interval algebra is, but the elements are points and so there are only three simple point relations: precedes, equals, and follows. As expected, only a fragment of the interval algebra can be translated to the point algebra, namely all constraints that represent convex sets of intervals. Vilain and Kautz show that a constraint propagation algorithm excluding the relation (A {precedes $\vee$ follows} B) exists, with time complexity $O(n^3)$, where n is the number of time points. Later, Van Beek [Beek 1989] designed a $O(n^4)$ algorithm that includes this relation. This was later improved in [Meiri 1990] to $O(n^3)$. Other similar algebras include subsets of interval and point algebra so that the consistency problem becomes polynomial. For example NB [Nebel and Burckert 1995] is an algebra that admits only a subset of all the possible disjunctions that can be formed with the 13 basic relationships and that is polynomial, and actually the maximal subclass that contains all the 13 basic relationships.

In quantitative temporal reasoning the constraints and temporal information can be metric. The simplest case is when this information is available in terms of specific dates or another precise numeric form and the durations and execution times of actions are absolute numeric values. Constant time algorithms can efficiently answer queries about occurrences by comparing these values. However, precise numeric information is not always available, in which case a constraint on the occurrence of an event E can be represented as an interval [*lower-bound, upper-bound*]. Similarly binary constraints between events take the form $X - Y \in [l, u]$, where X and Y are events (time-points). This representation first appeared in [Dean and McDermott 1987] and then formalized and generalized in [Dechter, Meiri et al. 1991] by applying techniques from the Constraint Satisfaction Problem (CSP). This formalization was named Simple Temporal Problem (STP), while the generalization of the constraints $X - Y \in [l, u]$ to constraints of the form $X - Y \in [l_1, u_1] \vee \ldots \vee [l_n, u_n]$ was named Temporal Constraint Satisfaction Problem (TCSP). The STP can be solved in polynomial time $O(n^3)$, where n is the number of time-points, by using all-pairs shortest paths algorithm, while there is no polynomial algorithm for the TCSP.

The TCSP constraints can also be written as $l_1 \leq X - Y \leq u_1 \vee \ldots \vee l_n \leq X - Y \leq u_n$. A generalization of this kind of constraints is when we allow constraints of the form $l_1 \leq X_1 - Y_1 \leq u_1 \vee \ldots \vee l_n \leq X_n - Y_n \leq u_n$, where the pairs X_i, Y_i can be distinct in each disjunct thus allowing the encoding of non-binary constraints. This class of problems is the Disjunctive Temporal Problem first introduced in [Stergiou and Koubarakis 1998] and it is very expressive. Other types of problems include STPs with strict inequalities [Gerevini and Cristani 1997] (e.g. $X < Y$), STPs with inequalities (e.g. $X \neq Y$) [Koubarakis 1992], STPs with conditional branches (CSTPs) [Tsamardinos, Pollack et al. 2000], and the temporal networks of Barber [Barber 2000], which extend TCSPs with forbidden non-binary assignments. This latter classes reaches the expressiveness of DTPs.

Another important class of problems is the Simple Temporal Problem with Uncertainty (STPU) [Vidal and Ghallab 1996]. All the above classes of problems consider consistency as the existence of an assignment to the temporal variables that respects all the constraints (except CSTPs and CDTPs, as we will see at Chapter 6). The STPU however, distinguishes between time-points that are assigned times by the deliberative agent (called controllables), and those that are assigned time by other agents, Nature, or other uncontrollable procedures. With STPUs there are many

notions of consistency (which in STPU terminology is called controllability): for example, whether there is a way to assign times to the controllable events so that they respect the constraints no matter how the when uncontrollable events occur.

We refer the reader to [Vila 1994; Gennari 1998; Schwalb 1998] for a more detailed discussion of all of the above temporal problems. For the purposes of this dissertation the most important class of problems are the DTPs, CSTPs, TCSPs, STPUs, and Barber's temporal networks, which will be explained in more detail later in this section and at Chapter 6.

An additional related area of research is that of Temporal Constraint Logic Programming, which involves embedding temporal models in the Constraint Logic Programming paradigm. Logic Programming in general is a class of languages in which information is described using "if-then" rules of assertion. Each rule is of the form H:- B_1 , …, B_n , where H is the head of the rule and B_1 , …, B_n are atoms that constitute the body. Prolog is the most popular of this type of languages. Constraint Logic Programming began as a natural merger of the two paradigms of CSP and Logic Programming [Smith 1995]. The combination helps make Constraint Logic Programming programs both expressive and flexible, and in some cases, more efficient than other kinds of programs[Hentenrick 1989; Jaffar and Lassez 1994]. A CLP program is composed of rules of the form H : - B_1 , …, B_n , C_1 , …, C_m where again H is the head of the rule, B_1 , …, B_n are the non-constraint atoms in the body of the rule, and C_1 , …, C_m are the constraint atoms in the body of the rule. Note that constraint atoms cannot appear in the head of the rule.

CLP programs differ from traditional logic programs in the way the constraint atoms are processed. In standard logic programming, the constraint and non-constraint atoms are treated equally. In CLP, specialized techniques for deciding which constraints are entailed, called constraint propagation algorithms, may be applied to the constraint atoms only. In Temporal Constraint Logic Programming the constraints allowed in the rules are temporal constraints. [Schwalb 1998] describes two such languages that combine the TCSP model to a couple of CLP languages. Our hope is that our DTP solving techniques of Chapter 4, like TCSP, will also find their way into Temporal Constraint Logic Programming languages.

2.3 The Simple Temporal Problem (STP)

We now formally and technically discuss a particular constraint-based temporal reasoning formalism called the Simple Temporal Problem.

A Simple Temporal Problem (***STP***) is a collection of variables V, each of which corresponds to a time point or instantaneous event, and a collection of constraints between the variables, C. The constrains are binary and each constraint between two variables X and Y is of the form $Y - X \leq b_{XY}$, where b is any real number and is called the ***bound*** of the constraint. The STP can be represented with a directed, weighted graph $<V, E>$, where each edge E has an associated bound: for each constraint $Y - X \leq b_{XY}$ in C there is an edge in E from X to Y with weight b_{XY}. The corresponding graph or network is called a Simple Temporal Network or ***STN*** and the two terms STP and STN can be used interchangeably; similarly the term constraint and edge will be used interchangeably. Figure 2-2 displays an example STN. A pair of constraints $Y - X \leq b_{XY}$ and $X - Y \leq b_{YX}$ can be grouped together in one constraint $-b_{YX} \leq Y - X \leq b_{XY}$ and be shown on an STN as an ***interval*** edge from X to Y with weight the *interval* $[-b_{YX}, b_{XY}]$. Intuitively such an edge can be

interpreted as the constraint that Y should only follow X after t time units, where t belongs in $[-b_{YX}, b_{XY}]$. Figure 2-2(b) shows the original STN with its edges converted to interval edges. Interval edges are frequently used in the literature as an intuitive way to draw and display STNs.

Definition 2-1: A ***Simple Temporal Problem*** (***STP***) is a directed edge-weighted graph $<V,E>$ where V is the set of temporal variables (nodes) and E the set of edges. The variables represent time-points and their domain is the set of reals $\Re$ (and so time is assumed dense). Each edge is labeled by a real value weight that is called the ***bound*** of the edge. The bound of the edge from node X to node Y is denoted as b_{XY} and the edge as XY or (X, Y). Each edge XY imposes the following constraint on the values assigned to its end-points: $Y - X \leq b_{XY}$.

The definition for STPs with edges annotated with an interval instead of a single bound is equivalent and we will use both interchangeably.

Definition 2-2: Let N be an STP and $V=\{p_1, ..., p_n\}$ be its set of nodes. A tuple $X=(x_1, ..., x_n)$ is called a ***solution*** *iff* the assignment $\{p_1 = x_1, ..., p_n = x_n\}$ satisfies all the constraints. The network N is ***consistent*** if and only if there is at least one solution. Two networks are ***equivalent*** if and only if they represent the same solution set.

Let p be any path in an STN between two nodes X and Y. Path p imposes the following constraints on the pair of nodes:

$$p_1 - X \leq b_{Xp_1}$$
$$p_2 - p_1 \leq b_{p_1 p_2}$$
$$\vdots$$
$$Y - p_n \leq b_{p_n Y}$$

where the p_i are the intermediate nodes in the path p from X to Y. By adding these inequalities, it is easy to see that p imposes (implies) the constraint:

$$Y - X \leq \sum_{i=1}^{n} b_{p_i p_{i+1}}$$

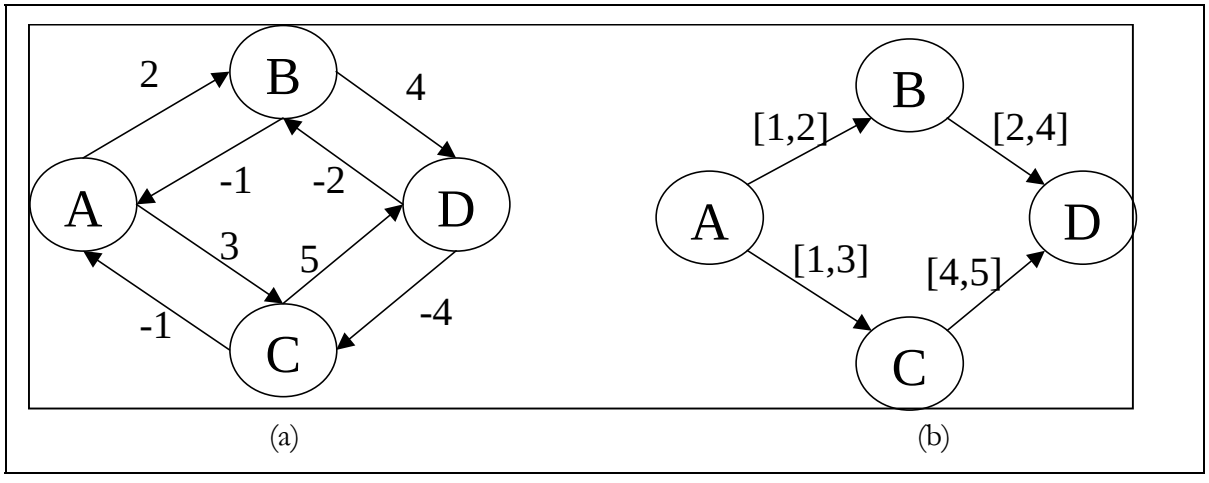


Figure 2-2: An example STP

where $X=p_1$ and $Y=p_n$. The tightest constraint between X and Y is obviously imposed by the ***shortest path*** between them. The weight of the shortest path is called the ***distance*** between the two nodes and is denoted as d_{XY}. In a consistent STN, the distance d_{XY} is the smallest real for which the constraint $Y - X \leq d_{XY}$ holds in all solutions. The constraint $Y - X \leq d_{XY}$ is an induced or implied constraint.

Theorem 2-1: An STP is consistent *iff* for every i , $d_{ii} \geq 0$ or equivalently, there are no negative cycles in the graph [Dechter, Meiri et al. 1991; Meiri 1992].

STPs have a very attractive property [Dechter, Meiri et al. 1991]: determining consistency is polynomial. All induced constraints between any pair of nodes X and Y, i.e. all distances, can be found by applying path-consistency (see Chapter 3), which is equivalent to running an all-pairs shortest paths algorithm on the graph [Cormen, Leiserson et al. 1990]. The network can then be solved (i.e. a consistent assignment to the variables can be found), without backtracking, in polynomial time. In [Chleq 1995; Cesta and Oddi 1996] two techniques for incrementally determining consistency in an STP are described. That is, given a consistent STP N and some additional constraints, the algorithms determine if the resulting STP N' remains consistent, in time typically much less than what it would take another consistency checking algorithm running from scratch on N'.

Definition 2-3: The ***distance array*** for an STP $N = <V,E>$ is the $|V| \times |V|$ array with elements a_{ij} : $a_{ij} = d_{ij}$, where d_{ij} is the distance between nodes i , $j \in V$. The array can be considered as representing a complete network where there is an edge (i, j) for every i , $j \in V$ with bound $b_{ij} = d_{ij}$, called the ***distance graph*** of the STP. The terms distance array and distance graph will be used interchangeably.

When we tighten the constraints between every pair of nodes X and Y of an STP so that the only admissible values for their difference $Y - X$ are values that participate in some solution, then the STP is minimal. Formally:

Definition 2-4[2]: An STP N is ***minimal*** if for each (induced or explicit) constraint $-b_{YX} \leq Y - X \leq b_{XY}$ and every $t \in [-b_{YX}, b_{XY}]$ there is a solution s such that $Y_s - X_s = t$, where Y_s , X_s are the times assigned to Y and X respectively in s.

Theorem 2-2: Given a consistent STP N the equivalent STP N' induced by its distance graph is the minimal STP equivalent to N. (Theorem 2.3, page 21, [Gennari 1998]).

Theorem 2-2 thus determines one way to calculate the minimal STP for a given STP N. Since the minimal STP is obviously unique, finding the minimal STP and the distance graph are equivalent operations. The theorem can also be used to construct a solution to the STP as follows:

1. Pick arbitrarily any time point X, and set $X = v$, where v is any arbitrary value (typically, $X=TR$, where TR is the time reference point defined below, and $v=0$)
2. Pick a node $Y \in V$, and assign time t to Y such that $t \in [-d_{YX}, d_{XY}]$
3. Re-calculate the minimal STP propagating the constraint $Y - X = t$

[2] Adopted from [Gennari 1998], Definition 1.0.4, page 8, for STPs.

4. Go to step 2 until all the nodes have been assigned a time.

The algorithm is correct because (i) if the network is consistent there will always be a value in $[-d_{YX}, d_{XY}]$ to assign in step 2, and (ii) by the theorem and the definition of the minimal STP, any value $t \in [-d_{YX}, d_{XY}]$ can be extended to a solution.

As we will see in the next subsection, STPs can be used to represent the constraints of temporal plans. When such a plan is executed (equivalently, its STP is executed), times are assigned to the nodes of the STP (in which case we say that they are executed), effectively dynamically constructing a solution to the STP. The above algorithm for constructing a solution requires the propagation of a constraint in step 3 each time a node is executed. When systems have to execute temporal plans under strict real-time constraints, it is uncertain how much time this propagation will require. In [Muscettola, Morris et al. 1998; Tsamardinos 1998; Tsamardinos, Morris et al. 1998] an STP is reformulated to an equivalent one such that step 3 in the algorithm above provably requires the minimum propagation. In particular, this reformulated equivalent STP has the minimal number of edges with the property that propagation of execution constraints at step 3 only needs to be propagated to the immediate neighbors of the executed node. This line of work on dispatching and executing temporal networks is reviewed again and extended for DTPs at Chapter 5.

Apart from representing and executing temporal plans, STPs can also be used to represent more general temporal information and can provide answers to the following types of queries:

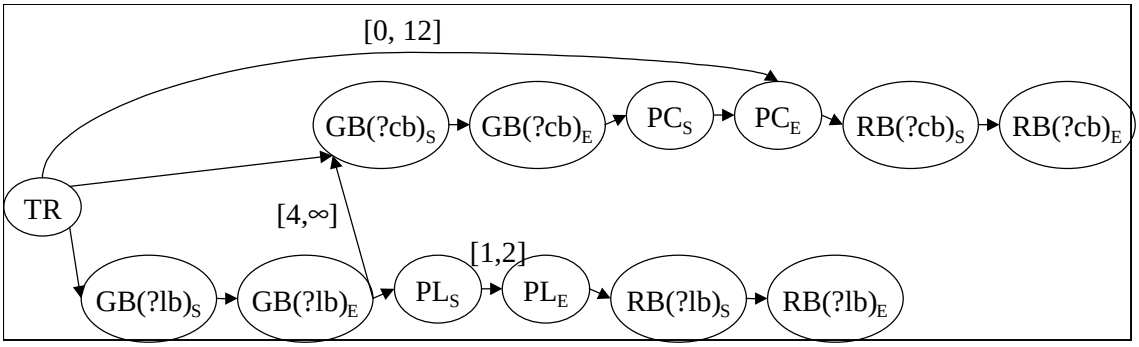


Figure 2-3: The STN for the painting example with additional constraints

- Is the information consistent? The answer is yes, if and only if, the corresponding STP is consistent.
- What is the allowable separation in the time between two events *X* and *Y*? The answer is *Y* can follow *X* by *t* time units, where $t \in [-d_{XY}, d_{YX}]$.

2.3.1 Using STPs to encode plans with quantitative temporal constraints

We consider a plan as consisting of at least a set of steps *S* and a set of temporal constraints *C* among the steps in *S*. A plan might contain other objects too, but sets *S* and *C* are what is minimally required for out purposes in this section. We construct an STP to represent a temporal plan in the following manner: For each step *A* in the plan, the nodes $\boldsymbol{A_S}$ and $\boldsymbol{A_E}$ are inserted in an STP; these correspond to the events of starting and ending the execution of the action respectively. To be able to encode absolute time constraints such as "the start of step *A* is at

9:00am, this Monday", a special node *TR* (the ***time reference point***) is inserted in the STP that represents the plan, and is arbitrarily related to a date and time, such as 12:00am 1/1/2000. All other time points of the STP should be constrained to occur after *TR*. An STP can represent the temporal aspects of a plan only if the constraints in *C* can be encoded by STP-like constraints, thus, for example, disjunctive constraints are not included.

We will now present an example (taken from [Yang 1997]): Figure 2-4 displays a plan for painting the ceiling and painting the ladder. There are eight steps in the plan, with Initial-State and Goal-State corresponding to two dummy steps of starting and ending the plan. All arrows indicate ordering constraints constraining the action at the end of the arrow to succeed the one at the beginning of the arrow. The argument to steps Get-Brush and Return-Brush is a paintbrush. In addition to the ordering constraints shown in Figure 2-2 let us assume that the temporal constraints should also hold:

- Painting the ceiling should end by 12:00pm 1/1/2000.
- Painting the ladder takes between 1 and 2 hours.
- Getting the ceiling brush (Get-Brush(?cb)) should occur at least 4 hours after (finishing) getting the ladder brush (Get-Brush(?lb))

The resulting STP that corresponds to the plan with the addition of the temporal constraints is shown at Figure 2-3, where *TR* is assumed to be 12:00am 1/1/2000. For readability, the interval annotation for all edges is omitted when it is [0, +∞], time units are assumed hours, and instead of the whole action name, only the acronym is used. The two dummy steps are also omitted from the figure.

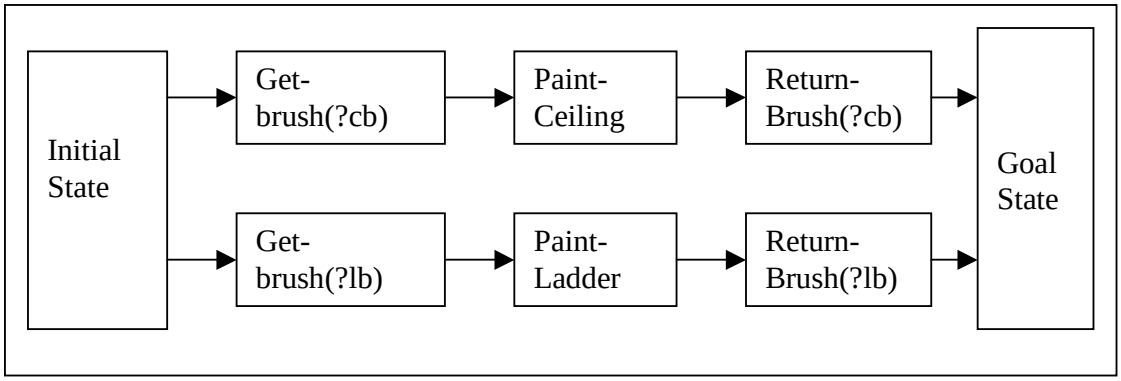


Figure 2-4: A simple temporal plan for painting a ladder and the ceiling

2.4 The Temporal Constraint Satisfaction Problem (TCSP)

A Temporal Constraint Satisfaction Problem (TCSP) [Dechter, Meiri et al. 1991] is a generalization of the STP.

Definition 2-5: A ***Temporal Constraint Satisfaction Problem*** (***TCSP***) is a directed edge-weighted graph $<V,E>$ where V is the set of temporal variables (nodes) and E the set of edges. The nodes represent time-points and their domain is the set of reals $\Re$. Each edge XY is labeled by a set of intervals $I_1 = [l_1 , u_1]$, …, $I_n = [l_1 , l_n]$ and corresponds to the constraint:

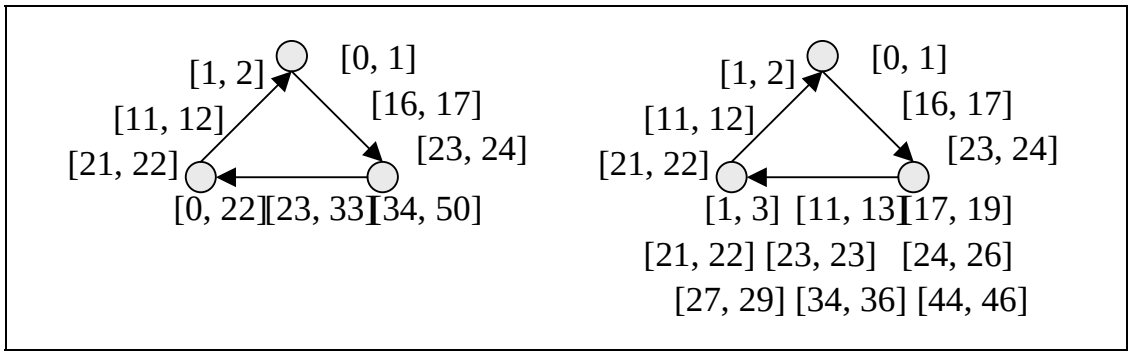


Figure 2-5: A TCSP and its minimal TCSP

$l_1 \le Y - X \le u_1 \ldots l_n \le Y - X \le u_n$ or equivalently $Y - X \in \cup_i I_i$

The definitions for the solution of a TCSP, TCSP consistency, and equivalent TCSPs are generalizations of the corresponding definitions for STPs. Notice than an STP is a TCSP where all edges are annotated with one interval. By selecting one interval I_i from each edge we form a component STP. The consistency of the STP can be determined in polynomial time and if there is a consistent component STP then the TCSP is also consistent since any solution of the STP is also a solution of the TCSP. Conversely, if there is no consistent projected STP, the TCSP is inconsistent. Unfortunately, there is an exponential number of possible component STPs in any TCSP and thus the problem of solving the TCSP is NP-Hard. This is proven by reduction to the 3-coloring problem in [Dechter, Meiri et al. 1991]:

Theorem 2-3: Consistency checking in TCSP is NP-Hard.

Similar to the minimal STP of a given STP, we can define the minimal TCSP as follows.

Definition 2-6: An TCSP N is ***minimal*** if for each (induced or explicit) constraint $Y - X \in \cup_i I_i$ and every $t \in \cup_i I_i$ there is a solution s such that $Y_s - X_s = t$, where Y_s, X_s are the times assigned to Y and X respectively in s.

The minimal TCSP is unique for a given TCSP (see page 19, [Gennari 1998]) but unfortunately the problem of calculating the minimal TCSP is exponential because of *fragmentation*: that is the number of intervals in the minimal TCSP is exponential to the number of intervals on the edges of the original TCSP (see Figure 2-5 for an example taken from [Schwalb 1998]). For state of the art algorithms on solving TCSPs see [Schwalb 1998].

Running a path-consistency algorithm on the network approximates calculating the minimal TCSP. A path-consistency algorithm is sound but incomplete, meaning that it can sometimes correctly determine that a TCSP is inconsistent, but it will not always identify a TCSP as inconsistent if this is the case. A complete algorithm for determining consistency requires backtracking search, meaning incrementally selecting one interval from each edge, thus gradually building a consistent component STP. When the search reaches a dead-end and there is no way to select an interval from an edge in any consistent way with the current STP, the search backtracks to alter a previous choice of an interval. In addition, Schwalb [Schwalb 1998] suggests that every time we select an interval from an edge to extend the currently built component STP, we propagate this decision by running path-consistency and tighten the values on the un-instantiated

edges. Schwalb suggests two looser types of path-consistency that avoid the fragmentation problem (but they do not have the pruning power of the full path consistency) and reports significant performance improvements.

2.5 The Simple Temporal Problem with Uncertainty (STPU)

The Simple Temporal Problem with Uncertainty (STPU) model extends the STP by taking into account the uncertain nature of durations of some tasks. It distinguishes between *contingent* constraints whose effective duration is observed only at execution time, and *controllable* ones, whose instantiation is controlled by the agent [Vidal 2000]. This contrasts the STP, TCSP, and most other such models that assume that times can be assigned to the variables (nodes) at will by the agent. In planning applications when these models encode temporal plans, some events are inherently uncontrollable and this is exactly what the STPU model is trying to take into account.

In [Vidal and Morris 2001] an STPU is defined as follows:

Definition 2-7: A ***Simple Temporal Problem with Uncertainty*** (***STPU***) is a tuple $<V, E, C>$, where V is a set of temporal variables (nodes), E a set of edges between the nodes associated with an interval $[l, u]$, and $C \subseteq E$, called *contingent edges or contingent links*, and each edge XY imposes the constraint $l \leq Y - X \leq u$ (equivalently $Y - X \in [l, u]$) . It is also required that for each contingent link with the interval $[l, u]$, $0 < l < u$.

The non-contingent links are called requirement links and when the distinction between contingent and requirement links is dropped it is easy to see that we end up with an STP. The semantics of an STPU is that it is Nature, or some other uncontrollable agent, that will instantiate the value of Y for every contingent link XY, respecting the constraint of the contingent link $l \leq Y - X \leq u$. Equivalently, Nature will pick the value t so that $t = Y - X \in [l, u]$, given the value the agent assigned to X. For example if X is a node that corresponds to starting an action A and Y corresponds to the event of ending the action A, and we know that A will take between 3 and 5 time units, we can model this fact with the contingent link $Y - X \in [3, 5]$. During execution, if action A starts at time unit 10, i.e. X=10, Nature will assign a time to Y so that $Y - X \in [3, 5]$, in other words Y can be anything in the interval [13, 15].

This distinction between contingent and requirement links gives rise to three notions of consistency, which for STPUs is instead called ***controllability***. Each notion is appropriate in a different execution situation. Before we define them, we need to first defined the following concepts [Vidal and Morris 2001]:

Definition 2-8: A ***projection*** of an STPU $\Gamma <V, E, C>$ is an STP derived from Γ by replacing each requirement link by an identical STP edge, and each contingent link XY with interval $[l, u]$ by an STP link with equal lower and upper bounds $[b, b]$ for some b such that $l \leq b \leq l$. .

Definition 2-9: Given an STPU $\Gamma <V, E, C>$ a ***schedule*** T is a mapping $T: N \rightarrow \Re$, where $T(x)$, $x \in V$ is called the ***time*** of time-point x. A schedule is consistent if it satisfies all the link constraints. From a schedule, we can determine the durations of all contingent links that finish prior to a time-point x. We will call this the ***prehistory*** of x with respect to T, denoted by $T_{<x}$.

Definition 2-10: An ***execution strategy*** S is a mapping $S: \Pi \rightarrow \Phi$, where Π is the set of all possible projections and Φ the set of all possible schedules. An execution strategy S is ***viable*** if $S(p)$ is consistent for each projection p.

Definition 2-11: An STPU is Weakly Controllable if there is a viable execution strategy, or in other words if every projection is consistent.

Definition 2-12: An STPU is Strongly Controllable if there is a viable execution strategy S such that $T_{p1}(x) = T_{p2}(x)$, where $T_{p1} = S(p1)$ and $T_{p2} = S(p2)$, for each time-point x and projections p_1 and p_2.

Thus, a "Strong" execution strategy assigns a fixed time to each executable time-point irrespective of the outcomes of the contingent links.

Definition 2-13: An STPU is Dynamically Controllable if there is a viable execution strategy S, such that if two schedules T_{p1} and T_{p2} where $T_{p1} = S(p1)$ and $T_{p2} = S(p2)$, have the same prehistory with respect to time-point x, then $T_{p1}(x) = T_{p2}(x)$, for each time-point x and projections p_1 and p_2 .

Thus a "Dynamic" execution strategy assigns a time to each executable time-point that may depend on the outcomes of the contingent links in the past, but not those in the future (or present). In other words, a dynamic execution strategy schedules the controllable events according to the history of the execution and the timing of the uncontrollable events, so that, no matter how the future uncontrollable events turn out to be, it can keep scheduling events respecting the constraints until it reaches a solution.

The STPU is reviewed here for its similarities with the Conditional Simple Temporal Problem (CSTP) formally defined at Chapter 7. The two classes of problems deal with two different sources of uncertainty and share analogous definitions and concepts of consistency and controllability. Unfortunately however, the algorithms and techniques used in for determining Strong, Weak, and Dynamic Controllability, cannot directly be applied to the equivalent notions in CSTPs. More information on STPU is found at [Vidal and Ghallab 1996; Morris and Muscettola 1999; Morris and Muscettola 2000; Vidal 2000; Vidal 2000; Vidal and Morris 2001; Vidal and Coradeschi August 1999]. As a final comment, note the important result that despite the fact that Weak-Controllability has been proven co-NP-complete [Morris and Muscettola 1999; Vidal and Fragier 1999], recently a polynomial algorithm was found to determine Dynamic Controllability [Vidal and Morris 2001].

3. CONSTRAINT SATISFACTION PROBLEMS

Our work contributes to solving temporal problems and makes no claims for original contributions in the area of solving general, non-temporal, Constraint Satisfaction Problems (***CSPs***). However, since a number of techniques and concepts we intend to use in subsequent chapters are directly related to Constraint Satisfaction a short background introduction is presented in this chapter. The CSP literature is large [Smith 1995; Dechter and Frost 1999; Frost October 1997] and this chapter certainly does not cover everything. It is intended to provide the reader with the minimum CSP background needed to understand the usage of these techniques in the later chapters.

3.1 Definition of Constraint Satisfaction Problems (CSP)

This section defines Constraint Satisfaction Problems and presents some examples.

Definition 3-1: A (***finite***) Constraint Satisfaction Problem (***CSP***) is a triplet $<V, C, D>$ where V is a set if variables $v_i \in V$ and D a set of finite domains $D(v_i)$, so that each v_i is allowed to take values from the domain $D(v_i) \in D$. The set of constraints C contains assignments of values to their variables (noted as $\{v_i \leftarrow val_1 , \ldots , v_n \leftarrow val_n \}$) that are considered illegal[3].

Constraint can also be represented in other ways that may be more convenient. For instance, if v_1, v_2, and v_3 each have a domain consisting of the integers between 1 and 10, a constraint between them might be the algebraic relationship $v_1 + v_2 + v_3 > 15$, implying that only triplets of values whose total is more than 15 are valid.

Definition 3-2: For the CSP $<V, C, D>$, a CSP assignment $\{v_1 \leftarrow d_1 , \ldots, v_n \leftarrow d_n\}$ (partial or complete) of the variables $v_i \in V$, where $d_i \in D(v_i)$ ***violates*** a constraint $C_j \in C$ if it is a superset of C_j . An assignment is ***consistent*** if it does not violate any constraint in C. A complete and consistent assignment is called a ***solution***. If there is a solution to a CSP then it is called ***consistent***..

Definition 3-3: A ***unary*** constraint is an illegal assignment that involves only one variable. A *binary* constraint is an illegal assignment that involves two variables. A ***binary*** *CSP* is one in which each constraint is unary or binary. An ***n-ary*** CSP is one in which the arity of some constraints is greater than two.

[3] More often, in the definition of CSPs, the constraints are defined as subsets of allowable assignments. Obviously, the two definitions are equivalent. The reason for departing from the standard definition is that constraints defined our way are the same as no-good assignments that will be used for no-good learning.

Example 3-1: Let $V = \{a, b, c\}$, $D(a)$=D(b)=D(c)={1, 2} and $C = \{C_1 = \{a \leftarrow 1, b \leftarrow 2\}$, $C_2 = \{a \leftarrow 2, c \leftarrow 2\}$, $C_3 = \{b \leftarrow 2, c \leftarrow 2\}$, $C_4 = \{a \leftarrow 1, c \leftarrow 1\}$, $C_5 = \{a \leftarrow 1, c \leftarrow 2\}\}$. There are three variables, five constraints, and two values in each variable domain. One solution to this CSP is $\{a \leftarrow 2, b \leftarrow 1, c \leftarrow 1\}$ and thus the CSP is consistent. The assignment $\{a \leftarrow 2, b \leftarrow 1\}$ is not a solution because it is not a complete assignment, while the assignment $\{a \leftarrow 2, b \leftarrow 1, c \leftarrow 2\}$ is not a solution because it violates the constraint $C_2 = \{a \leftarrow 2, c \leftarrow 2\}$.

Typical examples of problems that map to CSPs are the N-Queens problem [Minton, Johnston et al. 1990], the SAT problem [Du, Gu et al. 1997], and the graph-coloring problem [Yang 1997] p. 58. In the N-Queens problem we seek an assignment of N queens on an N×N chessboard, in such a way that no queen threatens any other queen. There are various ways to represent this problem as a CSP and numerous studies have demonstrated the implications of alternative representations on solution efficiency. In one typical representation there are n variables v_i , $I=1,\ldots,n$ each with domain set {1, … , N} representing the column (or row) to assign to the queen. In the final solution, each queen will be placed at position (i, v_i). The constraints in the CSP allow only placements in which the queens do not threaten each other.

In the SAT problem we seek for a truth assignment to a set of variables so that a propositional logic formula over those variables is satisfied. If the formula is in Conjunctive Normal Form, each clause forms a constraint to be satisfied in a CSP problem in which the variables correspond to the variables in the logic formula.

Finally, in the graph-coloring problem, we would like to color each vertex of a graph with a different color than any other vertex connected to. Each vertex to be colored forms a variable in a corresponding CSP, each possible color for the vertex, a value in its domain, and each arc between vertexes induce a constraint specifying that the colors for the two vertexes should be different.

3.2 Solving Constraint Satisfaction Problems

The literature on solving CSPs is vast and for the most part considers binary CSPs. In general, CSPs are solved by search and deduction or a combination of the two. In addition, there are two types of searches, systematic search and stochastic (local) search. Chronological backtracking is an example of systematic search (shown below). As the name implies, systematic search searches the whole space of possible assignments either explicitly or implicitly. Research in this area consists of finding algorithms that do not repeat unnecessary search effort with the least possible amount of bookkeeping. Some of these algorithms are Dynamic Backtracking [Ginsberg 1993], Conflict Directed Backjumping (***CDB***) [Prosser 1993; Chen and Beek 2001] and no-good recording [Dechter 1990], all described in detail in the survey paper [Dechter and Frost 1999]. Stochastic search methods move in a hill-climbing manner in the space of complete instantiations. The disadvantage of this kind of approach is that it does not report an inconsistency if the problem has no solution and this is the reason they will not be considered in this dissertation[4].

A generate-and-test is a very simple search method that could be used to solve CSPs. Each possible assignment is first generated and then tested against each constraint to ensure it does not

[4] There are some exceptions to the rule however (see [Ginsberg and McAllester 1994]).

violate it. A slightly more intelligent approach is to use a backtracking Depth-First search. During search an assignment for the variables is built incrementally. As soon as the current assignment violates a constraint there is no need to keep extending it, as every extension will still violate the same constraint. Therefore, the search underneath that subtree is stopped and the search tries a different value for the last variable. When all values of the last variable have been tried, the search backtracks to the previous variable. This kind of search is also called ***chronological backtracking***. Finding one solution to the CSP of Example 3-1 is shown at Figure 3-1. At the bottom of each leaf representing an inconsistent assignment, one constraint that is violated by the leaf's assignment is shown. In this dissertation, we will only concern ourselves with finding one solution to a CSP or proving the CSP has no solutions, and we will not be concerned with the problem of finding all solutions of a CSP, even though most times extending the algorithms to report all solutions is trivial (see [Chen and Beek 2001] pp. 59-60 for extending CDB to find all solutions). In general, finding a solution to a CSP is NP-complete.

Deduction in the CSP framework is known as constraint propagation or consistency enforcing. Deduction algorithms reason with constraints to prune out inconsistent values or partial assignments either before start searching for a solution, i.e. as a preprocessing step, or during search. The better known of these techniques include ***node-consistency***, ***arc-consistency***, and ***path-consistency*** [Montanari 1974; Mackworth 1977; Tsang 1993]. A CSP is *k-consistent* iff given

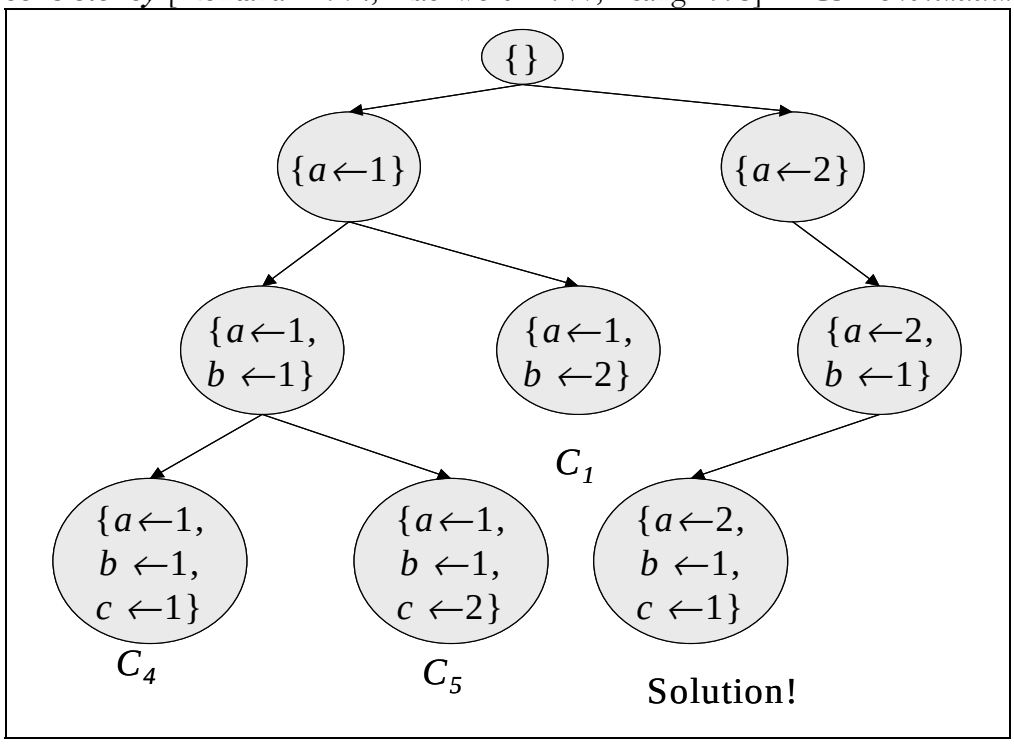


Figure 3-1: Chronological search for solving the CSP of Example 3-1.

any consistent partial instantiation of any $k-1$ variables, there exists a consistent instantiation of any kth additional variable. For binary CSPs, the terms node-, arc-, and path-consistency correspond to 1-, 2-, and 3-consistency respectively.

Structure-driven algorithms, which attempt to characterize the topology of constraint problems that are tractable, take advantage of the special structure of some CSP problems, or transform general CSPs to CSPs with special topology that makes it easy for algorithms to solve. Structure-driven algorithms will also be not considered in this dissertation because they depend on the special structure of problems, and there is no indication that the CSP-type problems we will be considering have any special structure or could be easily converted to problems with special structure. In particular when converting a DTP to a meta-CSP (explained later), the constraints are neither binary (a typical assumption of structure-driven algorithms; see [Bessiere 1999] for a discussion on the thesis that non-binary constraints have not achieved enough focus in the literature) nor they are represented explicitly (instead they are discovered during search). It is not obvious how structure-driven algorithms could be extended to take into account implicit constraints.

3.3 A Theory of No-goods

This section illustrates the concept of no-goods [Dechter 1990; Frost and Dechter 1994; Ginsberg and McAllester 1994; Schiex and Verfaillie 1994; Schiex and Verfaillie 1994; Yokoo 1994; Dechter and Frost 1999], which is an advanced and powerful tool in solving CSPs and Dynamic CSPs (to be introduced later).

Definition 3-4: A constraint A is induced by a CSP $<V, C, D>$ if it is satisfied (not violated) in all the CSP solutions.

Let $A \downarrow V$ denote the assignment that results from projecting assignment A on the variables of V. For example, if $A = \{a \leftarrow 2, b \leftarrow 1, c \leftarrow 2\}$ and $V = \{a, c\}$ then $A \downarrow V = \{a \leftarrow 2, c \leftarrow 2\}$. In addition, let $C_V \in C$ be the constraints in C that involve only the variables in V, and V_C be the variables in V that appear in the constraints in C.

Definition 3-5: A ***no-good*** of the CSP $<V, C, D>$ is a pair $<A, J>$, where A is an assignment of values to a subset of V, and $J \subseteq V$ such that no solution of the CSP $<V, C_J, D>$ contains A. Stated otherwise, the constraint forbidding A is induced by the CSP $<V, C_J, D>$. The set J is called the no-good justification (or culprit).

Example 3-2: Consider again the CSP of Example 3-1 restated here for clarity: $V = \{a, b, c\}$, $D(a) = D(b) = D(c) = \{1, 2\}$ and $C = \{C_1 = \{a \leftarrow 1, b \leftarrow 2\}, C_2 = \{a \leftarrow 2, c \leftarrow 2\}, C_3 = \{b \leftarrow 2, c \leftarrow 2\}, C_4 = \{a \leftarrow 1, c \leftarrow 1\}, C_5 = \{a \leftarrow 1, c \leftarrow 2\}\}$. Each constraint C_i trivially defines a no-good: For example, C_1 implies that $<C_1, \{a, b\}>$ is a no-good. Now notice that when $a \leftarrow 1$ constraint C_4 precludes c from taking value 1 and C_5 precludes c from taking value 2 either. Since these are the only values in the domain of c, $D(c)$, we can infer that $a \leftarrow 1$ is an induced constraint. Thus, the pair $<A, J>$, where $A=\{a \leftarrow 1\}$, is a no-good. What is the justification J? The constraints that imply the no-good are C_4 and C_5. Thus, the variables that "justify" the no-good are the variables of these two constraints and so $J = \{a, c\}$. $C_J = \{C_2, C_4, C_5\}$ and, as the definition requires, the CSP $<V, C_J, D>$ implies that A cannot

be contained in any solution of it[5]. Notice at this point, that a no-good $<A, J>$ does not only depend on the constraints C of the CSP, but also on the domains in D. If the domain of c in this example contained more values than 1 and 2 we could not have inferred $A=\{a \leftarrow 1\}$ is an induced constraint.

Theorem 3-1: Let $<A, J>$ be a no-good, and $<A', J'>$ be such that $J \subseteq J'$ *and* $A \subseteq A'$ (**noted as $<A, J> \leq <A', J'>$**). Then $< A', J'>$ is a no-good. (Theorem 3.1 in [Schiex and Verfaillie 1994], page 5)

Example 3-3: We will use again the CSP of **Example 3-2** to illustrate the theorem. We showed that $<\{a \leftarrow 1\}, \{a, c\}>$ is a no-good. By the theorem, $<\{a \leftarrow 1, c \leftarrow 2\}, \{a, c\}>$, $<\{a \leftarrow 1\}, \{a, b, c\}>$, and $<\{a \leftarrow 1, c \leftarrow 2\}, \{a, b, c\}>$ are all no-goods too.

Obviously, ≤-minimal no-goods are preferred since they implicitly define more no-goods according to Theorem 3-1.

Theorem 3-2: Let $<A, J>$ be a no-good, then $<A{\downarrow}J, J>$ is a no-good (theorem 3.2 in [Schiex and Verfaillie 1994], page 5).

Intuitively, the theorem states that we can reduce the assignment of a no-good, by only considering the variables in the justification. For example, if $<\{a \leftarrow 1, c \leftarrow 2\}, \{a, b\}>$ is a no-good, then $<\{a \leftarrow 1, c \leftarrow 2\}{\downarrow}\{a, b\}, \{a, b\}> = <\{a \leftarrow 1\}, \{a, b\}>$ is also a no-good.

Theorem 3-3: Let A be a (partial) assignment of the variables in V, v_s be an unassigned variable in V, and $\{A_1, \ldots, A_m\}$ be all the possible extensions of A along v_s, using every possible value of $D(v_s)$. If $<A_1, J_1>, \ldots, <A_m, J_m>$ are no-goods, then $<A, \cup_i J_i>$ is a no-good (corollary 3.1 in [Schiex and Verfaillie 1994], page 5)

Example 3-4: Theorem 3-3 is exactly what we used intuitively in Example 3-2 to infer that **$<\{a \leftarrow 1\}, \{a, c\}>$** is a no-good. Let us illustrate now the same CSP and the same derivation again in light of Theorem 3-3. If we let $A = \{a \leftarrow 1\}$, we see that $A_1 = \{a \leftarrow 1, c \leftarrow 1\}$ and $A_2 = \{a \leftarrow 1, c \leftarrow 2\}$ are all the possible extensions of A along variable c. Trivially (see the discussion in Example 3-2), $<\{a \leftarrow 1, c \leftarrow 1\}, \{a, c\}>$ and $<\{a \leftarrow 1, c \leftarrow 2\}, \{a, c\}>$ are no-goods, or equivalently $<A_1, \{a, c\}>$ and $<A_2, \{a, c\}>$ are no-goods. By the theorem we infer that $<A, \{a, c\}$ is a no-good too.

No-goods will be mostly used for pruning the search space. If for example we know that $<A, J>$ is a no-good, then we should not search in any superset of A since it provably will not lead to a solution. Therefore, no-goods with smaller sets A are preferable since they prune larger parts of the search space. In Example 3-4, by using Theorem 3-3**,** we inferred the *smaller* no-good $<\{a$

[5] The reader might be confused at this point why we define a no-good as the pair $<A, J>$, where the justification J is the set of the *variables* involved in the constraints that imply A, instead of having J to be the set of the actual constraints. Indeed this is the approach taken at [Schiex and Verfaillie 1994]. In the previous example that means that the no-good we discovered would not be $<\{a \leftarrow 1\}, \{a, c\}>$ but instead $<\{a \leftarrow 1\}, \{C_4, C_5\}>$. Notice that C_J is a superset of $\{C_4, C_5\}$ and thus Definition 3-5 does not provide as specific justifications. However, when employing no-goods for solving the Disjunctive Temporal Problem (see chapter 4) it is more convenient to encode and use as justifications sets of variables than sets of constraints. The two definitions of no-goods are equivalent for all purposes of this dissertation. For a more thorough discussion on the subject see [Richards 1998] pp. 54-55.

$\leftarrow 1\}$, $\{a, c\}>$ from the no-goods $<\{a \leftarrow 1, c \leftarrow 1\}$, $\{a, c\}>$ and $<\{a \leftarrow 1, c \leftarrow 2\}$, $\{a, c\}>$. Theorem 3-2 and Theorem 3-3 will be the main tools in building smaller and smaller no-goods during search in a never ending attempt to reduce the required search.

3.4 A No-good Learning Algorithm

This section presents a no-good learning algorithm published in [Schiex and Verfaillie 1994]. The algorithm utilizes Theorem 3-2 and Theorem 3-3 to successively build smaller and smaller no-goods from previously identified ones and is presented in Figure 3-2. The top procedure (line 1) **NR** should be called with a partial assignment A and the set of still unassigned variables U. So, for any CSP $<V, C, D>$, the initial call to **NR** should then be **NR**($\emptyset$, V). If a solution is found to the CSP by **NR** it is reported at line 3, otherwise a justification J for the failure of an extension to assignment A is returned. The justification J is the same as a no-good justification: a set of variables whose interrelated constraints are responsible for the failure to extend the assignment A. In other words, if **NR**(A, U) returns J then $<A, J>$ is a no-good. During the search and the recursive calls, **NR** records the no-goods it discovers and uses them to prune the search space.

Let us trace the algorithm in Figure 3-2. If there are no more variables to assign, that is $U=\emptyset$, then the assignment A is a solution and so we and stop and report it (line 2). Otherwise, a variable x is chosen by function **select-variable** from the unassigned variables to be assigned a value (line 5). At line 6, all possible values v to x are assigned in turn. The function **backward-check** called with an assignment A' tests whether A' violates an existing (explicitly given) constraint or a recorded no-good, in other words if it is inconsistent. If A' is consistent, **backward-check** returns the empty set. If not, it returns a justification that depends on the kind of the failure: if a recorded no-good $<A", J>$ is a superset of A' then the returned justification is J . If on the other hand, an explicit constraint C_i is violated the justification returned is the variables that appear in the violated constraint C_i .

There are three distinct paths of execution in the code in Figure 3-2:

1. $K = \emptyset$ and $x \in J$-sons
2. $K = \emptyset$ and $x \notin J$-sons
3. K not equal to $\emptyset$

We will show that no matter what path of execution is followed in any loop iteration, NR will return the correct justification. Let us assume for a moment that in every loop iteration only case (3) occurs, i.e. every extension of A in the call of **NR** along the variable x violates some induced or explicit constraint (and so the code never enters line 10). Let us denote with v_m the value of variable v in the figure above at the mth iteration of the loop at line 6. Then, as described above, **backward-check** returns a justification K_m so that $<A \cup \{x \leftarrow v_m\}, K_m>$ is a no-good. By Theorem 3-3 $<A , \cup_m K_m>$ is a no-good and the justification $\cup_m K_m$ is exactly what is stored at variable J in line 18, recorded in line 21, and eventually returned, in line 22.

Let us relax now the assumption that in every iteration we fall into case (3) and assume that we either fall into case (3) or (1). Whenever we fall into case (1) then **NR** is called recursively at line 10. If it ever returns, then because of line 3, a solution has not been found. **NR** will return with the justification of the failure J-sons which implies that $<A \cup \{x \leftarrow v_m\}, J\text{-sons}_m>$ is a no-good. As before $<A, J>$ is a no-good where $J=\cup_m J_m \cup_r J\text{-sons}_r$ is the variable returned at line 22.

Finally, if in any iteration we fall into case (2) then we stop iterating (by setting *BJ* to *True*) and return *J* = *J-sons*. Stopping the iteration means that we are not trying any other values for *x*. Let us assume that we do not distinguish this case from when *x* is involved in *J*-sons$_k$ (case 1). Then, the next iteration of the loop will assign a different value to *x* and so the recursive call to **NR** will be with the assignment $A \cup \{x \leftarrow v_{m+1}\}$. Since *x* is not involved in the justification *J*-sons$_k$ that means that changing its value will have no effect to the recursive call which will return again with a failure and the same justification. Since trying different values for *x* has no effect, we can ***backjump*** variable *x* without trying the rest of its values. This is why the algorithm at line 14 handles

```
1. NR(A, U)
2. If U=∅ Then
3.       A is a solution, Stop and report A.
4. Else
5.       x = select-variable(U), J =∅, BJ = false
6.       For each v ∈ D(v) until BJ
7.              A' = A ∪ {x ← v },
8.              K = backward-check(A')
9.              If K = ∅                ;; That is, A can be extended along x
10.                    J-sons = NR(A', U – {x} )
11.                    If x ∈ J-sons Then
12.                           J = J ∪ J-sons
13.                    Else
14.                           J =J-sons
15.                           BJ=true
16.                    Endif
17.             Else
18.                    J = J ∪ K
19.             Endif
20.      EndFor
21.      If BJ = false then record(project(A, J), J) Endif
22.      Return J
23. Endif
```

Figure 3-2: The no-good recording algorithm

differently this case and sets *BJ* to be *True* so that we exit the loop and return from the call to **NR**. This type of backjumping is closely related to ***conflict-directed backjumping* (*CDB*)** proposed in [Prosser 1993] .

The actual no-good recording occurs at line 21. As already mentioned, *J* contains the justification why $<A, J>$ is a no-good, but from Theorem 3-2 we also know that $<A \downarrow J, J>$ is also a (potentially smaller) no-good and so it is preferable to record the latter one. Function **project** (A, J) returns $A \downarrow J$ and function **record(***A, J*) records the no-good $<A, J>$. Notice, that if

we have to backjump and so *BJ* is true, then there is no need to record the no-good. It will have already been recorded at the recursive call to **NB**.

The algorithm is proved to be sound, complete, and terminating at [Schiex and Verfaillie 1994].

3.4.1 An example run of the no-good learning algorithm

This subsection presents an example taken from [Schiex and Verfaillie 1994] to clarify the workings of the algorithm in Figure 3-2. The algorithm will run on the problem presented at Figure 3-3. In the CSP of the figure there are five variables x_1-x_5 corresponding to each circle with domains as given. The constraints among the variables are binary. For example, the edge from x_1 to x_3 represents the constraining assignments between these two variables, shown beside it. All the ordered pairs of values in the array are forbidden assignments, e.g. { X1 ← a , X3 ← a} is a constraint.

Calling the function **NR** of Figure 3-2 and using the variable ordering (x_1 , x_2 , x_3 , x_4 , x_5) will create the search tree of Figure 3-4. Simple chronological backtracking algorithm explores the full tree while the NR algorithm only expands the bold edges. The no-goods built *and recorded* are given inside the ovals with a dotted arrow giving the nodes where they have been built (trivial no-goods corresponding to each leaf marked with × are not shown in the figure as they are not recorded). Let us see how the first recorded no-good $<\{x_2 \leftarrow a\}, \{x_2, x_3\}>$ is built:

1. the assignment $A_1 = \{x_1 \leftarrow a, x_2 \leftarrow a , x_3 \leftarrow a\}$ is inconsistent because it violates the constraint $\{x_2 \leftarrow a , x_3 \leftarrow a\}$ (check the array by the edge x_3 to x_2 at Figure 3-4. Trivially then $<A_1 , \{x_2 , x_3\}>$ is a no-good and function **backward-check** at line 7 returns $\{x_2 , x_3\}$.
2. in the next loop iteration, the assignment $A_2 = \{x_1 \leftarrow a, x_2 \leftarrow a , x_3 \leftarrow b\}$ violates again a constraint between variables x_2 and x_3 and again $<A_2 , \{x_2, x_3\}>$ is a no-good and **backward-check** returns $\{x_2 , x_3\}$.
3. it is now possible to apply Theorem 3-3 (line 21) and infer that $<\{x_1 \leftarrow a, x_2 \leftarrow a\}, \{x_2, x_3\}>$ is a no-good. Before recorded however, the call to **project** (line 21 again) reduces the no-good to $<\{x_2 \leftarrow a\}, \{x_2, x_3\}>$.

One may notice that the empty 0-ary constraint is induced last. This constraint simply states that the CSP is inconsistent and its justification $\{x_3 , x_4 , x_5\}$ is a minimal justification of the CSP inconsistency.

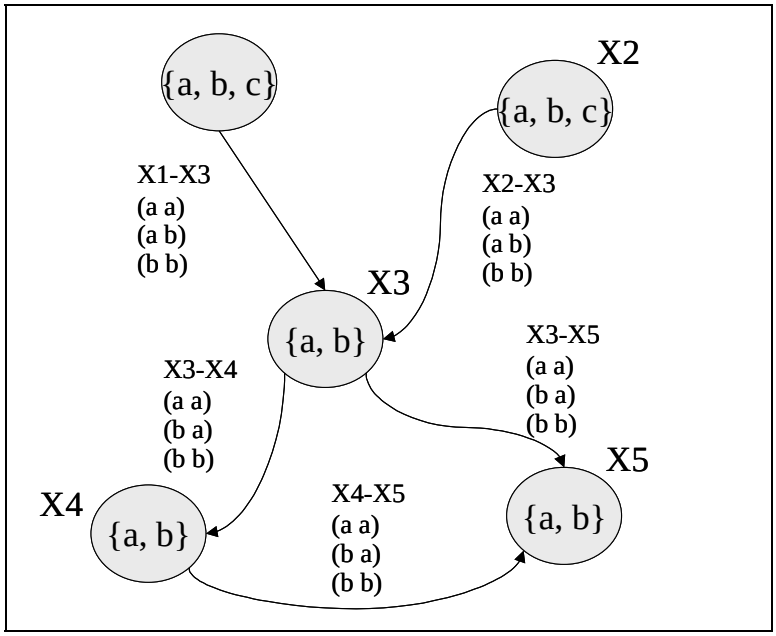


Figure 3-3: An example CSP

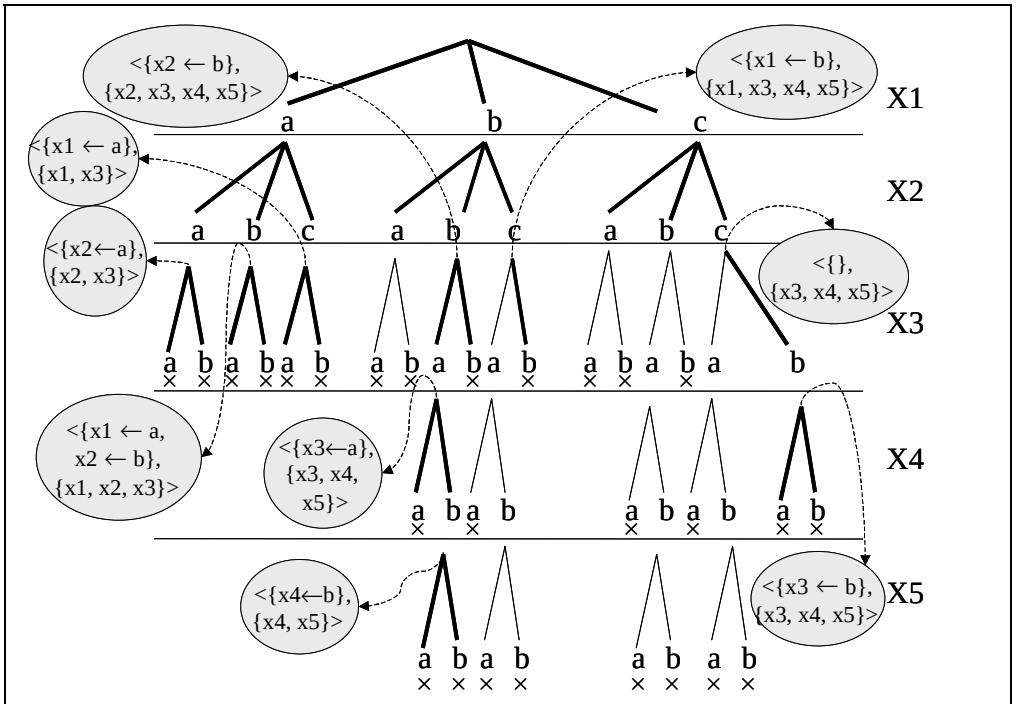


Figure 3-4: The search tree created by backtrack and no-good recording algorithms

3.5 Forward Checking and No-goods

This section describes the CSP technique called forward-checking [Haralick and Elliot 1980; Nadel 1989; Tsang 1993] and shows how it can be combined with the no-good recording algorithm of Figure 3-2.

In forward-checking when an assignment A is extended along a variable x to provide a new partial assignment $A' = A \cup \{x \leftarrow v\}$, the values of the unassigned variables that are inconsistent with the new assignment A' are removed from the corresponding domains. Since the domains of the variables change dynamically in forward-checking, we will denote with $D(u)$ the original domain of a variable and with $d_A(u)$ the current domain of u under assignment A (and drop the index A when it is implied by the context). Whenever for some unassigned variable u the domain $d_A(u)$ is reduced to the empty set, the partial assignment A' cannot be extended to a solution and thus a different value v' should be tried for x, if there is such, or the search should backtrack. The main idea of forward-checking is to propagate the implications of the new assignment $\{x \leftarrow v\}$ and typically this results in early pruning. The savings are often greater than the overhead of the propagation and this is why forward-checking is very commonly used to solve CSPs.

The forward-checking algorithm as described above performs only a limited amount of

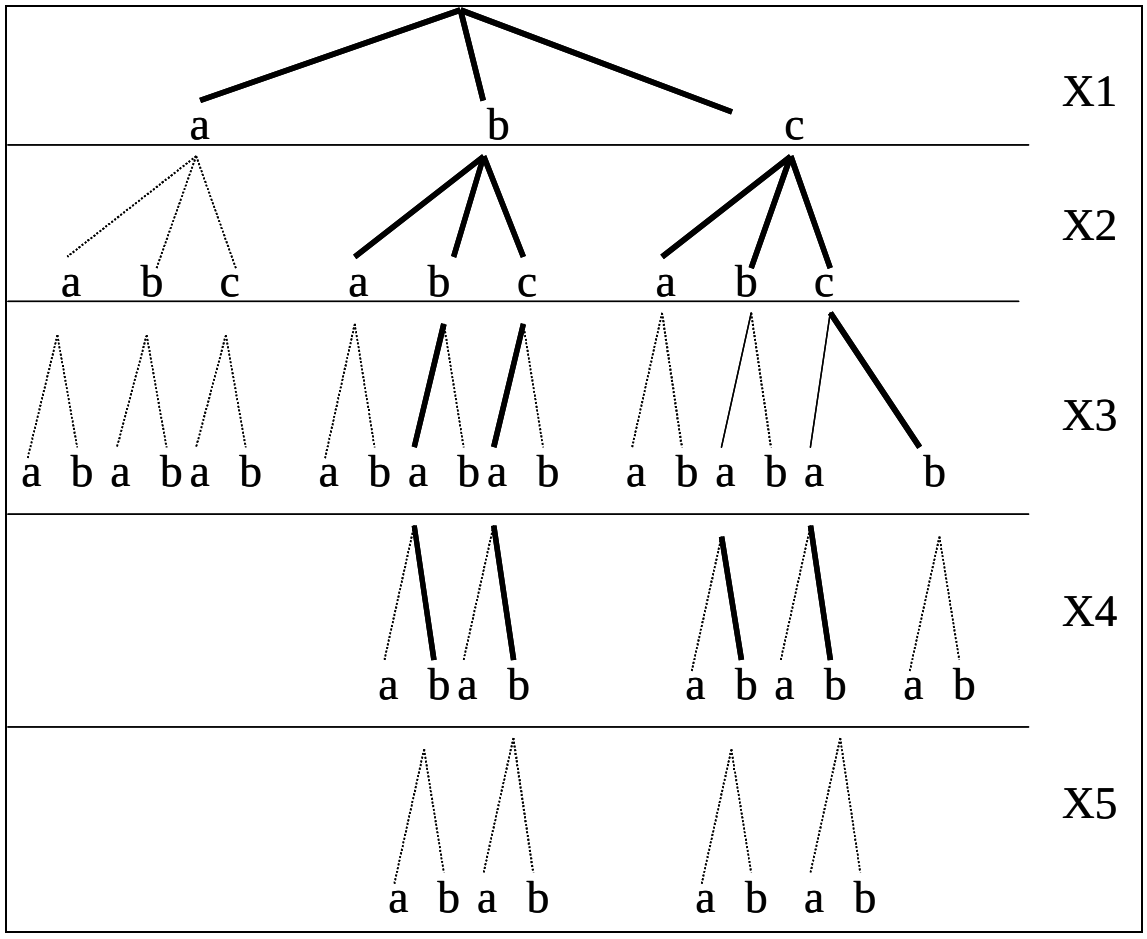


Figure 3-5: The search tree of the forward-checking with chronological backtracking algorithm.

propagation of the new assignment $\{x \leftarrow v\}$, namely removing inconsistent values from the domains of the unassigned variables. In general, removing a value from some domain may allow us to iterate and remove some other value; such propagation is captured in algorithms such as arc-consistency or path-consistency, and so on. We will not consider these more general cases of forward-checking here but will restrict ourselves to the simple form as described. Forward-checking can be used with a large number of backtracking mechanisms, including chronological backtracking and conflict-directed backjumping; most importantly for our purposes it can also be used with no-good recording, as it will be shown below.

Figure 3-5 above shows the search tree of the forward-checking algorithm with chronological backtracking. Chronological backtracking expands the whole tree, while the addition of forward-checking results in the expansion of only the nodes with the bold edges. To get an understanding of how forward-checking works, consider the first node where $A_1 = \{x_1 \leftarrow a\}$. Forward-checking this value removes from the domain of variable x_3 both values a and b, so $d_{A1}(x_3)$ becomes empty and thus there is no more search underneath the subtree for A_1 . After this failed attempt, x_1 becomes unassigned again, the domain of x_3 is restored to what it was before, and x_1 is subsequently assigned the next available value to produce the next assignment $A_2 = \{x_1 \leftarrow b\}$. This removes the value b from $d_{A2}(x_3)$ but since value a still belongs in $d_{A2}(x_3)$ no domain is reduced to the empty set and the search continues. When A_2 is extended to $A_3 = \{x_1 \leftarrow b, x_2 \leftarrow a\}$, value a is also removed by $d_{A3}(x_3)$ leaving no more values in that domain and the next value of x_2 is tried.

Notice that in this example, the value assignment $\{x_1 \leftarrow b\}$ is responsible for removing b from $d_{A3}(x_3)$ and the value assignment $\{x_2 \leftarrow a\}$ for removing a from $d_{A3}(x_3)$. The variables whose value assignments are responsible for removing some value from a domain $d_A(x)$ are the ***value killers*** of the domain. So, x_1 and x_2 are the value killers of $d_A(x_3)$.

3.5.1 Extending no-good recording with forward-checking

To incorporate forward-checking into the no-good recording algorithm, when building no-goods, each time a constraint is forward-checked and suppresses a value from a variable domain, it will be noted as a value killer of the domain. When a domain becomes empty, we may build a no-good composed of the current assignment and the set of all the value killers of this domain. For example, if assignment A suppresses all the values of a variable domain, we infer the no-good $<A, K>$, where K are the value killers of the empty domain.

The pseudo-code of the algorithm is given Figure 3-6 below. The functions **forward-check** and **un-forward** are introduced. Function **forward-check** (Figure 3-7) performs forward-checking. If the domain of a given variable is reduced to the empty set, the forward-checking stops and the set of all the *value-killers* of the variable is returned by calling **killers** (Figure 3-7), else the empty set is returned. Function **un-forward** is used upon backtracking: it restores the modified domains.

There are two noticeable differences however from the **NR** algorithm. One is in the possible use of the no-goods that are directly built when a failure occurs. When relying on backward checking, such no-goods are uninteresting since they only express the fact that the current assignment cannot lead to a solution because of the constraint it violates, and this fact is already

```
1. NR-FC(A, U)
2. If U=∅ Then
3.      A is a solution, Stop
4. Else
5.      x be a variable in U, J =∅, BJ = false
6.      For each v ∈ d_A(x) until BJ
7.              A' = A ∪ {x ← v }, K = forward-check(A', U)
8.              If K = ∅
9.                      J-sons = NR-FC(A', U – {x} )
10.                     If x ∈ J-sons Then
11.                             J = J ∪ J-sons
12.                     Else
13.                             J = J-sons, BJ=true
14.                     Endif
15.             Else
16.                     J = J ∪ K
17.                     record(project(A', K), K)
18.             Endif
19.             un-forward(x)
20.     EndFor
21.     If BJ = false Then
22.             For each v ∈ D(x) – d_A(x)
23.                     J = J ∪ killers-of-value(A, x, v)
24.             EndFor
25.             record(project(A, J), J)
26.     Endif
26.     Return J
23. Endif
```

Figure 3-6: The no-good recording algorithm extended with forward-checking [Schiex and Verfaillie 1994].

explicit in the constraint violated itself. With forward-checking, failure occurs when a domain is reduced to the empty set because of the conjunction of all the *value-killers* of the domain, a fact that is usually not explicitly stated anywhere in the CSP. Therefore, no-good recording with forward-checking may learn interesting data on leaves (i.e. when the code does not reach the recursive call at line 9) and this explains the use of the **record** function at line 17.

The second difference is subtle and lies in line 6. Notice that the loop in the **NR** (line 6) runs through all values in the original domain $D(x)$, while **NR-FC** iterates over the values of $d_A(x)$. This is because the values that do not belong in $d_A(x)$ have already been proved inconsistent with the current assignment by forward-checking. However, we can only apply Theorem 3-3 to build a new no-good $<A, J>$, if it is the case that for all values v_k in $\boldsymbol{D(x)}$, $<A \cup \{x \leftarrow v_k\}, J_k>$ is a no-good

(in which case $J = \cup_K J_k$) . The code up to line 20 calculates this union only over the variables v_k in $d_A(x)$. This is why lines 22-24 are required to guarantee to union over all values in $D(x)$[6].

```
forward-check (A, U)
1.  For each variable x in U
2.      For each value v in d_A(x)
3.          If A ∪ { v ← c } is inconsistent
4.              d_A(v) = d_A(v) – {c}
5.              If d_A(U) = ∅
6.                  Return killers-of-var (A, x)
7.              EndIf
8.          EndIf
9.      EndFor
10. EndFor
11. Return ∅

killers-of-var(A, x)
12. K = ∅
13. For each value v in d_A(x)
14.     K = K ∪ killers-of-value(A, x, v)
15. EndFor
16. Return K

killers-of-value(A, x, v)
17. If A ∪ { x ← v } violates a constraint C
19.     Return {var | var appears in C}
20. Else If A ∪ { v ← c } is a superset of no-good <A', J>
21.     Return J
22. EndIf
```

Figure 3-7: Forward-check for the no-good recording algorithm.

3.6 Dynamic CSPs and No-good Recording

A special kind of CSPs that deserves mentioning is Dynamic CSPs (***DCSP***) [Schiex and Verfaillie 1994]. A DCSP is a series of CPSs in which a constraint or variable is removed or added between successive elements of the series[7].

[6] The additional lines 22-24 are not in the original paper [Schiex and Verfaillie]. In direct communication with the first author of the paper it was established that lines 22-24 are indeed required for the algorithm to be complete. The experimental results in Ibid. are not invalidated however because lines 22-24 were included in the implementation and were only missing from the pseudo-code description of the algorithm. Careful re-implementation of the NR-FC algorithm also revealed a typo in the original publication of the algorithm. Line 17 appears as record(project(A, K), K) while it should be record(project(A', K), K).

[7] There is a slight confusion in the literature regarding DCSPs emanating from the fact that there is an alternative definition [Mittal and Falkenhainer 1990] where variables and constraints are allowed to be added to a CSP during search. In this definition, we will not concern ourselves with this other kind of DCSPs.

Definition 3-6: A ***Dynamic CSP*** **(DCSP)** P is a sequence $P_0, \ldots, P_i, \ldots$ of static CSPs, each one resulting from a change in the preceding one [Dechter and Dechter 1998]. This change may be a restriction (the imposition of a new constraint) or a relaxation (the removal of a constraint).

Obviously, each successive CSP could be solved with general CSP algorithms, however, there are algorithms that take into consideration that the new CSP shares structure with the previous one in the series, and reuse the computation that was performed for solving this previous CSP instance. In addition, extra requirements may be posed on the desirable solution for the current CSP, e.g. it might be preferable to find the solution that mostly resembles (i.e. requires the minimum number of changes from) the solution to the previous CSP.

One way to reuse computation is, once P_i is solved, to utilize the recorded no-goods: a no-good $<A, J>$ of P_i is also a no-good of P_{i+1} if no constraint in the variables in J has been removed. Therefore, before starting the search for a solution to P_{i+1} we can preprocess the no-goods for P_i and keep those that are still valid in P_{i+1} .The search in P_{i+1} will not have to rediscover them and can employ them for pruning. Experimental results in [Schiex and Verfaillie 1994] show that there are significant gains using this method. As an example, consider the case in which a relaxation of a constraint between variables x_m and x_k is the only difference between P_i and P_{i+1} . Suppose further that P_i has been proven inconsistent and NR-FC has returned the no-good $<\{\}, J\}$. If x_m and x_k are not members of J, then P_{i+1} is also inconsistent and this can be determined without any search.

The concept of no-goods is essential for the TPM solver that we have developed and will be discussed in detail later. Similar to DCSPs we can define Dynamic DTPs that will be used for modeling successive plan merging problems for example and use the same technique of preprocessing and subsequently utilizing the recorded no-goods from the previous DTP.

4. THE DISJUNCTIVE TEMPORAL PROBLEM

This chapter will present the Disjunctive Temporal Problem (**DTP**) in detail. *The main result of this chapter is an algorithm for checking the consistency of DTPs that is about two orders of magnitude faster than the previous state of the art algorithm on benchmark problems containing randomly generated DTPs. The algorithm has been fully implemented and tested in a system called* ***Epilitis*** (from the Greek word Επιλυτής = Solver; pronounced Epeeleetees with the accent at the last syllable) *that provides flexibility to experiment with alternative solving strategies.*

4.1 The Disjunctive Temporal Problem (DTP)

4.1.1 Definition of the Disjunctive Temporal Problem

The Disjunctive Temporal Problem (hereafter **DTP**) is a very expressive temporal reasoning formalism that subsumes a number of temporal problems in planning and scheduling. In particular, it subsumes the Simple Temporal Problem and Temporal Constraint Satisfaction Problem (STP and TCSP respectively) in [Dechter, Meiri et al. 1991] upon which it is based.

Definition 4-1: A Disjunctive Temporal Problem (***DTP***) is a pair $<V, C>$ where V is a set of temporal variables and C is a set of constraints among the variables. Every constraint $C_i \in C$ is of the form:

$$c_{i1} \vee \ldots \vee c_{in}$$

where in turn, each c_{ij} is of the form $x - y \leq b$, where c_{ij} is called the jth *disjunct* of the ith constraint and $x,y \in V$ and $b \in \Re$.

Definition 4-2: A (complete) ***assignment*** to the DTP variables V is a set of statements $x \leftarrow t$ assigning a time $t \in \Re$ to (all) variables in V.

Definition 4-3: An ***exact solution*** to a DTP is an assignment to all the DTP variables that respects all the constraints. A DTP is ***consistent*** if it has at least one exact solution; otherwise it is called ***inconsistent***.

Example 4-1: Consider the DTP $D = <V, C>$ where**:**

$$V = \{x, y, z, w\} \text{ and}$$

$$C = \{ \quad \{x - y \leq 5 \vee z - w \leq 10\},$$

$$\{y - x \leq -10 \vee w - z \leq -6\},$$

$$\{z - y \leq 5 \vee x - w \leq 10\}\}$$

D has four variables and three constraints among them. It is consistent: one exact solution is { $x \leftarrow 4, y \leftarrow 0, z \leftarrow 14, w \leftarrow 6$}. For this solution, the disjuncts c_{11}: $x - y \leq 5$, c_{22}: $w - z \leq -6$, and c_{32}: $x - w \leq 10$ are satisfied.

As seen in this example, in order to satisfy all the constraints in a DTP, it suffices to satisfy one disjunct from each constraint. A set containing one disjunct per constraint defines an STP since each disjunct is an STP-like constraint. In the above example, the disjuncts c_{11}, c_{22} , and c_{32} define an STP *S* and the solution presented is also a solution of *S*.

Definition 4-4 A (***partial***) ***STP component*** *S* of the DTP *D* is defined by picking a disjunct c_{ij} from (some) each disjunction C_i of *D*. A ***consistent component S*** of the DTP *D* is also called a ***solution*** of *D*.

The reason a consistent component *S* of *D* is called a solution of *D* is because every exact solution to *S* is also an exact solution to *D*:

Theorem 4-1: Every exact solution *A* to a consistent component *S* of the DTP *D* is also an exact solution of *D*.

Proof: Since *A* satisfies all dijsuncts c_{ij} of *S*, it satisfies all constraints C_i of *D*. ♦

Corollary 4-1: A DTP *D* is consistent if and only if there is a consistent component *S* of *D*.

Proof: If there is a consistent component *S* of *D*, then *S* has a solution *A* (since it is consistent) and thus by the previous theorem *A* is also an exact solution of *D* and so it is consistent. The converse is also true. If *A* is an exact solution to *D*, then *A* satisfies at least one disjunct c_{ij} of each C_i . This set $\{c_{ij}\}$ form by definition a component *S* of *D*. Since *A* satisfies all of them, it is a solution to *S*, and thus *S* is consistent. ♦

The connection between exact DTP solutions and consistent STP components can also be seen by rewriting the constraints of the DTP (Definition 4-1) from Conjunctive Normal Form (***CNF***) to Disjunctive Normal Form (***DNF***), i.e. converting the following formula:

$$\begin{aligned}
&(c_{11} \vee \ldots \vee c_{1n_1}) \wedge \\
&(c_{21} \vee \ldots \vee c_{2n_2}) \wedge \\
&\vdots \\
&(c_{m1} \vee \ldots \vee c_{mn_k})
\end{aligned}$$

by distributing the disjunctions over the conjunctions, to this new formula:

$$
\begin{aligned}
&(c_{11} \wedge c_{21} \ldots \wedge c_{m1}) \vee \\
&(c_{11} \wedge \ldots \wedge c_{m2}) \vee \\
&\vdots \\
&(c_{1n_1} \wedge \ldots \wedge c_{mn_k})
\end{aligned}
$$

Each disjunct in the latter formula defines a component STP S of D. D is a disjunction of a large number of STPs; a number exponential to the number of disjuncts per constraint. D is consistent if and only if any of the STPs is consistent.

The main problem addressed in this chapter is determining the consistency of a DTP. Consistency could be computed by finding one exact solution to the DTP. Indeed, this is the approach typically taken in the scheduling community when faced with similar problems involving quantitative constraints with a large number of choices for satisfying them. However, in the temporal reasoning community, a different approach has been explored. Instead of attempting to find an exact solution to a DTP D, DTP solvers look for consistent STP components. As already mentioned, any exact solution to these STPs is also an exact solution to the DTP. Therefore, a consistent component S, when discovered, provides not with just one exact solution as the former approach, but with a whole set of exact solutions: the set of solutions of S. Constructing an exact solution given S is computationally easy and can be done in $O(n^3)$ time, where n is the number of variables (see Chapter 2). Having more than one exact solution available is especially important to the planning community, since agents facing uncertainty can benefit from the scheduling flexibility. By carrying around S and dynamically executing it (i.e. constructing an exact solution at execution time depending on the circumstances), agents can handle uncertainty far more robustly than by predetermining an exact fixed schedule of execution.

The difference in the two approaches can also be stated as follows. Typically (with some exemptions, see [Cheng and Smith 1995; Cheng and Smith 1995]), schedulers attempting to solve problems with temporal constraints with n variables search in $\Re^n$ for an exact solution to the constraints, i.e. they search in the space of all possible assignments of reals to the variables. On the other hand discovering a consistent STP component of a DTP can be seen as a *search in the set* $\{c_{11}, c_{12}, \ldots, c_{1k}\} \times \ldots \times \{c_{m1}, c_{m2}, \ldots, c_{mk}\}$*, which is the set of all component STPs*, but can also be seen as the set of all possible choices of which disjuncts to satisfy.

Solving a DTP will mean identifying a solution (not exact solution) to it. Indeed, all previous DTP solvers [Stergiou and Koubarakis 1998; Armando, Castellini et al. 1999; Oddi and Cesta 2000] as well as the one described in this dissertation follow this approach. Searching for a solution in the space $\{c_{11}, c_{12}, \ldots, c_{1k}\} \times \ldots \times \{c_{m1}, c_{m2}, \ldots, c_{mk}\}$ can be cast as a meta-***CSP*** (Constraint Satisfaction Problem), as explained in the next section.

4.2 Solving Disjunctive Temporal Problems

The process of finding a solution STP S to a DTP D can be modeled as a CSP. Because the original DTP D is itself a CSP problem[8], we will refer to the component-STP extraction problem as the ***meta-CSP*** problem. The meta-CSP contains one variable for each constraint C_i in the DTP; the constraint will be denoted with the same symbol C_i. The domain of C_i is the set of disjuncts in the original constraint C_i. The constraints in the meta-CSP are not given explicitly, but must be inferred: an assignment satisfies the meta-CSP constraints iff the assignment corresponds to a component STP that is consistent. For instance, if the variable C_i is assigned the value $x - y \leq 5$ it would be inconsistent to extend that assignment so that some other variable C_j is assigned the value $y - x \leq -6$. The discussion is pictorially summarized in the table below:

	Original CSP	**Meta-CSP**
Variables	$x, y, z \ldots$	One variable C_i for each constraint C_i of the original DTP
Domains	$(-\infty, +\infty)$ for all variables	$D(C_i) = \{c_{i1}, \ldots, c_{im}\}$
Constraints	$x_1 - y_1 \leq b_1 \vee \ldots \vee x_n - y_n \leq b_n$ i.e $c_{i1} \vee \ldots \vee c_{in}$	Implicitly defined by the underlying semantics of the values.

Table 4-1: Correspondence between the DTP and the meta-CSP.

Definition 4-5: The meta-CSP $<V', C', D'>$ of a DTP $<V, C>$ is defined by $V' = \{C_i \mid C_i \in C\}$, $D' = \{ D(C_i) = \{c_{i1}, \ldots, c_{im}\}$, where each c_{ij} is a disjunct in C_i $\}$, and $C' = \{A$, where A is any assignment $\{C_i \leftarrow c_{ij}, \ldots, C_k \leftarrow c_{kl}\}$ such that $c_{ij}, \ldots, c_{kl}$ form an inconsistent STP$\}$.

From now on, we will use the term ***variable*** for a meta-CSP variable, that is a DTP constraint C_i, the term ***time-point*** for a DTP variable x, the term ***constraint*** or ***value*** for an STP-like constraint of a disjunct c_{ij}, and the term ***node*** for a CSP tree search node.

A straightforward forward-checking CSP algorithm can be used to solve DTPs (see Figure 4-1). The algorithm takes two parameters: A, denoting the set of already assigned variables and their assigned values and U, the set of as-yet unassigned variables. The initial call to solve a DTP $<V, C>$ should be made with $A = \varnothing$ and $U = C$, and the initial current domains $d(C)$ should be initialized to be the original domains $D(C)$. The function **select-variable** is a heuristic function that selects the next variable to assign, while **forward-check**(A, U) performs forward checking, i.e., it removes from the domains of the variables still in U all those values that are inconsistent with the current assignment A returning *false* if, as a result, one or more variables in U has a domain reduced to $\varnothing$. Function **forward-check** uses function **STP-consistency-check** which takes as input an STP and returns *true* if it is consistent and *false* otherwise. Finally, recall from Chapter 3 that function **un-forward** restores the domains of the variables to those before the last call to **forward-check**.

[8] However, when the variables can take real numbers it is not a *finite* CSP.

An example trace of the algorithm is in Figure 4-2 and Figure 4-3. On the left the search of the meta-CSP is shown, while on the right we display the STP entailed by the current assignment. The arcs marked by an "X" correspond to values removed by **forward-check**. Let us assume the order of selection of the variables is $C_1, \ldots, C_5$. First the algorithm picks C_1 and assigns it the one and only value in its domain c_{11}. This causes c_{21} to be removed from the domain of C_2 by **forward-check** (Figure 4-2(a)). Then, C_2 is assigned c_{21}. The new STP is inconsistent with value c_{42}, which is removed by **forward-check** (Figure 4-2 (b)). The next two value assignments are $C_3 \leftarrow c_{31}$ (Figure 4-3(c)) and then $C_4 \leftarrow c_{41}$ (Figure 4-3(d)). At this point, both values of C_5 are inconsistent with the current STP. Thus, the loop at line 3 will try the next value of $d(C_4)$. Chronological backtracking next attempts another value for C_4, but there are no possibilities since c_{42} has already been removed by **forward-check**. Thus, the search backtracks to C_3 and, since again there is no other choice to try for it, backtracks again to C_2 at Figure 4-3(e) (the "unfilled" nodes here are nodes that have already been explored). The next value for C_2 is c_{23} since c_{22} has been removed. The new value assignment $C_2 \leftarrow c_{23}$ causes the removal of both values in the domain of C_5 and since there are no more choices for either C_2 or C_1 the search ends in failure.

Notice that there is no need to check the consistency of the assignment A' at line 4 in the

```
Simple-DTP(A, U)
1.    If U=∅ stop and report A as a solution.
2.    C ← select-variable(U), U'←U-{C}
3.    For each value c of d(C) in some order
4.          A'=A∪{C←c}
5.          If forward-check(A', U')
6.                Simple-DTP(A', U')
7.          EndIf
8.          un-forward(U')
9.    EndFor
10.   Return failure

forward-check(A, U)
11.   For each variable C in U
12.         For each value c in d(C)
13.         If not STP-consistency-check(A ∪ {C←c})
14.               Remove c from d(C)
15.               If d(C) = ∅
16.                     return false
17.               EndIf
18.         EndIf
19.   EndFor
20.   Return true
```

Figure 4-1: **The simple-DTP algorithm**

algorithm. $A'=A\cup\{C\leftarrow c\}$ will always be consistent, because if $\{C\leftarrow c\}$ was inconsistent with A, then it would have been removed by **forward-check**.

In the next few sections we will present a few methods for improving the simple DTP solver of Figure 4-1. First we will show how to speed up forward checking of values. Then, three pruning methods are presented, i.e. removal of subsumed variables (RS), conflict-directed backjumping (CDB), and semantic branching (SB). The idea in RS is that we remove a variable from the set of unassigned variables U if the current STP already satisfies a disjunct of the constraint. In CDB, we identify the reasons why the current STP cannot be extended to a solution and backjump over a constraint if it does not participate in the justification set of constraints. In SB we use the underlying semantics of the CSP values, i.e. the fact that they are STP constraints, to infer new

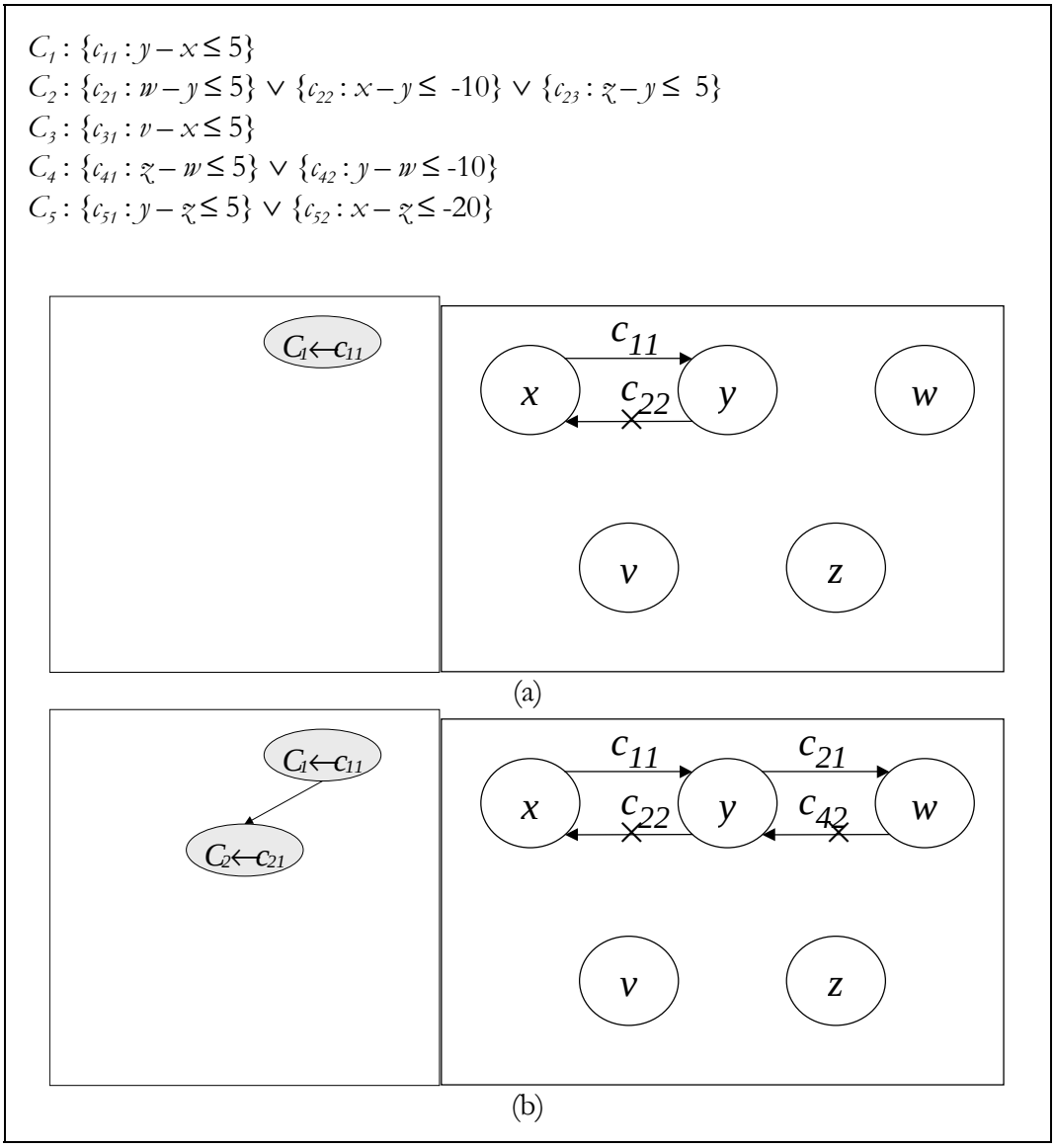


Figure 4-2: A trace of the Simple-DTP on an example

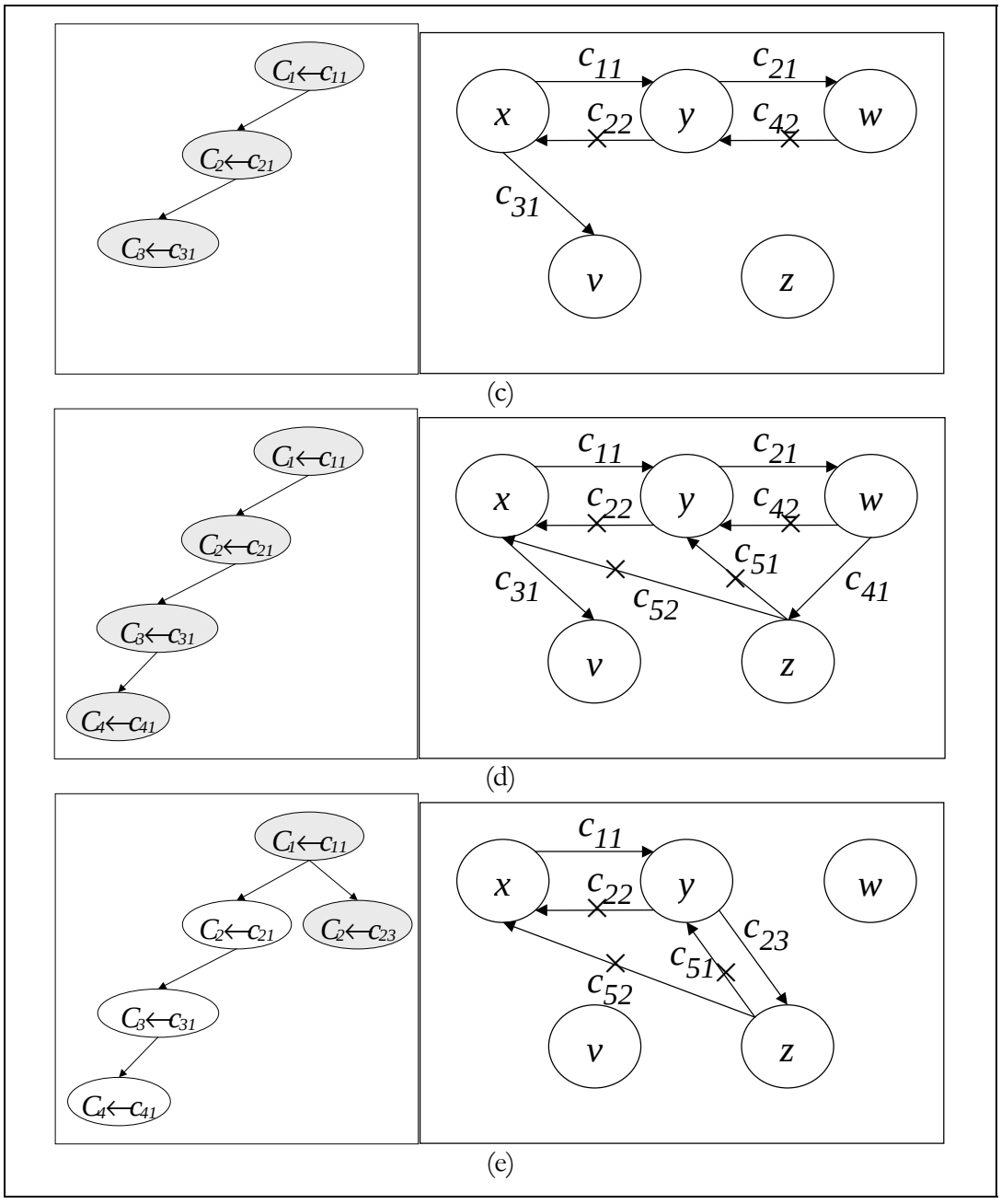


Figure 4-3: A trace of the Simple-DTP on an example

constrains that have to hold. We also present the equivalent of FC-off in SAT and a way for reducing the amount of forward checking that needs to be done.

4.2.1 Improving the Simple-DTP algorithm

A large part of the computation in algorithm **Simple-DTP** is spent at line 13 in **forward-check,** where each value of each variable is added to the set of constraints of the current assignment A and checked for STP-consistency. STP-consistency checking takes time $O(|V|^3)$ where $|V|$ is the number of time-points (and so forward-check takes time $O(v|V|^3)$, where v is the

number of values to forward check). Fortunately, there is a computationally less costly way of achieving forward checking of values, based on the following theorem.

Theorem 4-2: A value $c_{ij} : y - x \leq b_{xy}$ in an STP is inconsistent with a consistent STP S (that is $S \cup c_{ij}$ is inconsistent) if and only if the following condition holds:

$$b_{xy} + d_{yx} < 0 \quad (\textbf{FC-condition})$$

where d_{yx} is the distance between nodes y and x in S.

Proof: Let p_{yx} be the shortest-path from y to x in S and thus the length of p_{yx} is d_{yx}. If the FC-condition holds, then the path $p_{yx} \cup (x, y)$ has length $b_{xy} + d_{yx}$ and so it forms a negative cycle making S inconsistent.

Conversely, let us suppose that the FC-condition is false and prove that $S' = S \cup c_{ij}$ will be consistent. We will prove this claim by contradiction, i.e. we will assume FC condition is false, S' is inconsistent, and will derive a contradiction. If S' is inconsistent then there is a negative cycle, and because S was consistent before we added c_{ij} , that means that the negative cycle involves the new constraint c_{ij} . Let us assume the negative cycle is $c_{ij} \cup p'_{yx}$, for some path p'_{yx} . So $b_{xy} + d'_{yx} \leq b_{xy} + length(p'_{yx}) < 0$, where d''_{yx} is the distance between y

```
Improved-DTP(A, U, S)
1.    If U=∅ stop and report A as a solution
2.    C ← select-variable(U), U'←U-{C}
3.    For each value c of d(C) in some order
4.        A'=A∪{C←c}
5.        S' = maintain-consistency(c, S)
6.        If forward-check(S', U')
7.            Improved-DTP(A', U')
8.        EndIf
9.        un-forward(U')
10.   EndFor
11.   Return failure

forward-check(S, U)
12.   For each variable C in U
13.       For each value c : x – y ≤ b_xy in d(C)
14.       If b_xy + distance (y, x, S)
15.           Remove c from d(C)
16.           If d(C) = ∅
17.               return false
18.           EndIf
19.       EndIf
20.   EndFor
21.   Return true
```

Figure 4-4: **The Improved-DTP algorithm**

and x in S' for some path p'', which is the shortest path from y to x (see Figure 4-2). Since the negative cycle is a simple cycle (no loops allowed), then edge (x, y) is not a member of p''. Therefore, the shortest path p'' from y to x in S' is does not contain the new constraint c_{ij} and thus distance from y to x in S' and S is the same: $d_{yx} = d'_{yx}$. By (1) above we get that $b_{xy} + d'_{yx} = b_{xy} + d_{yx} < 0$, i.e. FC-condition holds, contrary to what we assumed.

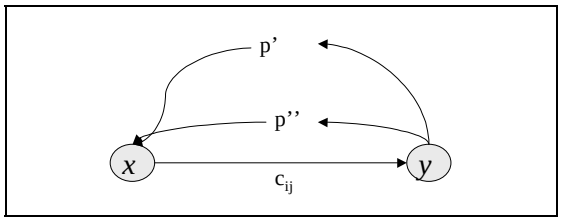


Figure 4-5: Proof of Theorem 4-2.

Theorem 4-2 indicates that to forward-check a particular value $y - x \leq b_{xy}$ against an assignment A we just need to check the FC-condition. In turn, this requires the calculation of the distances d_{yx} in STP S for all nodes x and y. One method for calculating all these distances efficiently is to calculate the distance array (also known as running full path consistency). Then each distance can be recovered with a simple matrix lookup at constant time. Running full path consistency takes time $O(|V|^3)$. After this step has been performed, **forward-check** takes time $O(v)$ where v is the number of values to be forward checked. Thus, in order to improve the performance of **forward-check** a synergy between the main algorithm and **forward-check** is required: at each node the distance array is calculated, and then it is used by **forward-check** to significantly improve performance.

Another technique would be to compute directional path consistency [Chleq 1995] where only a number of shortest paths is cached and the rest of the distances can be recovered in time at most $O(|V|)$.

Figure 4-4 shows the Simple-DTP with the above improvements. The main difference is that in each search node, i.e. at each call to **Improved-DTP**, an STP S corresponding to the current assignment A is maintained. Each time a new variable is assigned a value $\{C \leftarrow c\}$, the constraint is propagated in S by function **maintain-consistency**(c, S). Then the call to **STP-check-consistency**, Figure 4-1, line 13, can be substituted with line 14 in Figure 4-4 that only checks the FC-condition. Function **distance**(y, x, S) returns the distance between y and x in STP S. Function **maintain-consistency** can be implemented with full path consistency, in which case **distance** is a simple array lookup taking O(1) time, or directional path consistency, in which case **distance** is the algorithm described in [Chleq 1995] and it takes $O(|V|)$ time.

Comparing the three approaches together, i.e. (i) using algorithm **Simple-DTP** (i.e. no **maintain-consistency**), (ii) using full path consistency for **maintain-consistency** and a distance array look up to check the FC-Condition, and (iii) using directional path consistency for **maintain-consistency** and the algorithm in [Chleq 1995] to check the FC-condition, it takes $O(v|V|^3)$, $O(|V|^3 + v)$, and $O(|V|^3 + v|V|)$ on each tree node respectively, where $|V|$ is the number of time-points and v is the number of values to forward-check. Thus, the worst-case comparison favors

maintaining full path consistency (i.e. the distance array). However, since assignment A is built incrementally by adding constraints on each new node, we can use incremental versions of these previous techniques to build S, namely ***incremental full path consistency*** **(IFPC)** [Mohr and Henderson 1986] and ***incremental directional path consistency*** **(IDPC)** [Chleq 1995]**.** The incremental versions drop the exponent in all the above complexities to quadratic and so (ii) takes time $O(|V|^2 + v)$ and (iii) $O(|V|^2+v|V|)$. The average case comparison cannot easily be resolved theoretically and further experiments are required to determine under which conditions each method is the best. For our solver we used method (ii) and maintain the distance array at every node.

4.3 Pruning Techniques for DTP Solvers

This section presents three methods for pruning the search space of the meta-CSP, namely ***Conflict-Directed Backjumping, Semantic Branching,*** and ***Removal of Subsumed Variables***. The idea in all of these pruning techniques is to utilize the underlying semantics of the values of the meta-CSP, namely the fact that they express constraints on some STP, to make inferences regarding the infeasibility of certain space regions.

4.3.1 Conflict-Directed Backjumping for DTPs

The user will recall, from Chapter 3, that the main in idea in *CDB* is that when a dead end is encountered during search, the search does not naively backtracks to the most recently assigned variable. Instead, it backtracks to the most recently assigned variable that is *related* to the failure. The variables that are unrelated to the failure are backjumped, since trying to assign different values for them will result in the same dead end. It is obvious that to implement CDB one should be able to identify the culprit of the failures, i.e. the variables that participate in the constraints that lead to the failure. In Chapter 3, one technique for calculating culprits was shown for a general (non-temporal) CSP: the no-good justification was the reason why the no-good assignment could not be extended to a solution.

In the context of DTPs, the following example illustrates CDB.

Example 4-2: Consider the following DTP:

$$C_1 : \{c_{11} : y - x \le 5\}$$
$$C_2 : \{c_{21} : w - y \le 5\} \vee \{c_{22} : x - y \le -10\} \vee \{c_{23} : z - y \le 5\}$$
$$C_3 : \{c_{31} : v - x \le 5\} \vee \{c_{32} : z - v \le 10\}$$
$$C_4 : \{c_{41} : z - w \le 5\} \vee \{c_{42} : y - w \le -10\}$$
$$C_5 : \{c_{51} : y - z \le 5\} \vee \{c_{52} : x - z \le -20\}$$

which is the same as the one shown on the top of Figure 4-2 with addition of one more value c_{32} for the constraint C_3 . Searching for a solution for this DTP using either chronological backtracking or CDB is the same as the search shown in Figure 4-2(a) and (b) and Figure 4-3(c) and (d). When search hits a dead end (Figure 4-3(d)), the chronological backtracking backtracks to variable C_3 and assigns it the next available value c_{32} (Figure 4-6(a)). Subsequently, the search

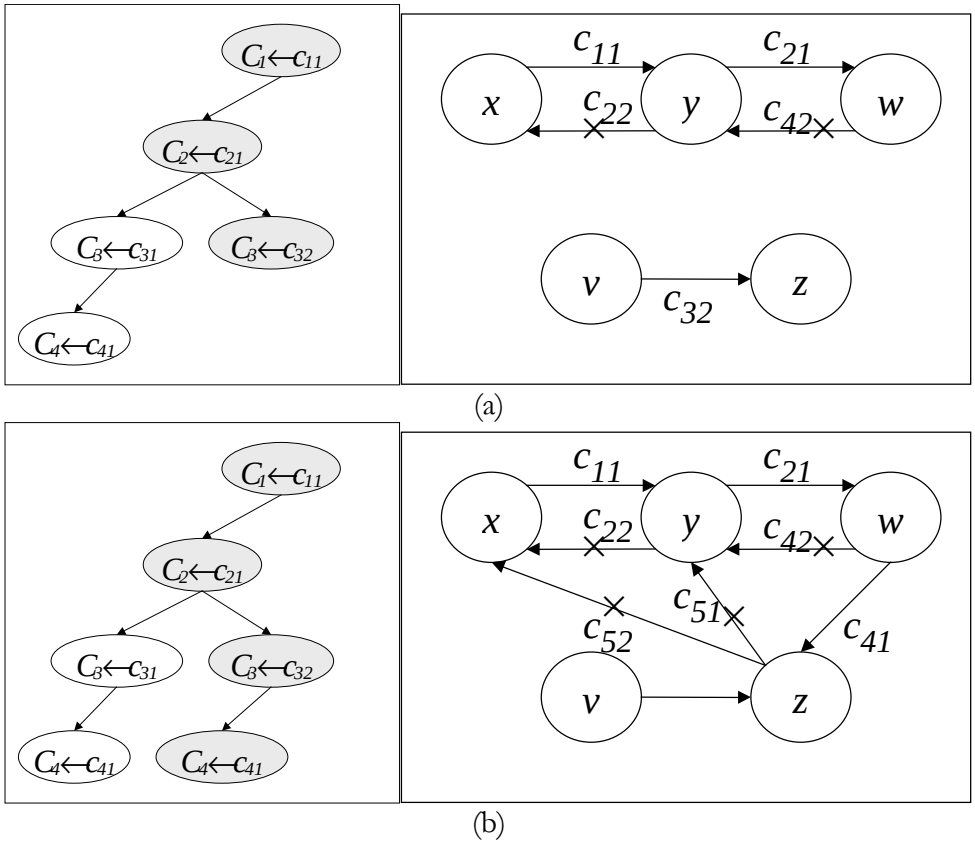


Figure 4-6: The chronological backtracking algorithm on the DTP of Example 4-2

continues at Figure 4-6 (b) with the value assignment $C_4 \leftarrow c_{41}$. The reason for backtracking in Figure 4-3(b) is that both values of C_5 , namely c_{51} and c_{52} are involved in a negative cycle with values c_{11} and c_{21} . This fact is still true when changing the value C_3 from c_{31} to c_{32} and thus we reach the "same" dead end in Figure 4-6(b). This observation allows CDB, which is a more intelligent backtracking method, to completely prune the search below the subtree at node $\{ C_2 \leftarrow c_{21} \}$ and backjump over variable C_3 (Figure 4-7) directly to C_2. The nodes crossed out in Figure 4-7 are the ones not explored by CDB but explored by chronological backtracking.

For CDB, it is necessary to be able to determine which variable-value assignments directly contribute to the failure, so that the irrelevant ones can be backjumped over. One method that has been proposed for this is in [Stergiou and Koubarakis 1998] and is named the ***dependency pointers scheme***. When the current assignment A is extended over a variable to $A'=A\cup\{C\leftarrow c\}$ forward checking is performed. If a value c' is removed from some domain, then obviously the

most recent value assignment $\{C \leftarrow c\}$ directly contributes to its removal and thus a dependency

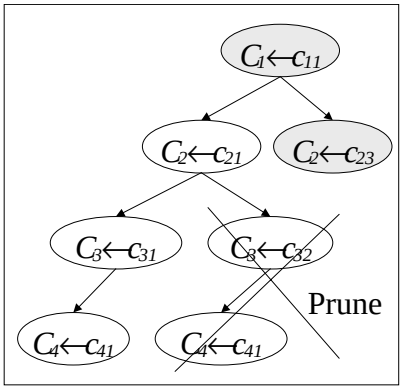


Figure 4-7: The complete search tree on the DTP of Example 4-2

pointer is stored from value c' to c. When search has to backtrack, the dependency pointers of the removed values of a domain that has been reduced to ∅ are checked, and the one that points to the most recently assigned variable is followed, backjumping over any irrelevant variables.

We will now present a new and different scheme for calculating the culprit of a failure that does not employ dependency pointers and additionally it integrates well with the semantic branching and no-good recording pruning techniques that are to be presented. Our technique will return the variables of the current assignment that are involved in the failure so that search can backjump to the most recent relevant variable. This return set of variables can also be used as the justification of no-goods in the no-good recording scheme for DTPs presented later.

When a dead end occurs, this is because **forward-check** has reduced a domain to the empty set, by removing every value c_j of that domain. This in turn implies that every c_i is part of a negative cycle p_i formed by constraints c_1 , …, c_k . For example, in Figure 4-6(b) both values c_{51} and c_{52} are removed from $d(C_5)$ because they form the negative cycles[9] $p_1 = (c_{11}$, c_{21} , $c_{51})$ and $p_2 = (c_{11}$, c_{21} , $c_{52})$. The variables that participate in the failure then are $\mathbf{vars}(p_1) \cup \mathbf{vars}(p_2) = \{C_1$, C_2 , $C_5\} \cup \{C_1$, C_2 , $C_5\} = \{C_1$, C_2 , $C_5\}$, where $\mathbf{vars}(p)$ are the variables whose value assignments are the constraints in p.

It becomes apparent that our technique requires the identification of a negative cycle for each removed value by forward check needs to be identified. This can be implemented by maintaining

justification-value$(c : y - x \leq b$, $S)$
1. $p =$ **shotest-path**(y, x, S)
2. Return **vars**$(p \cup c\}$

Figure 4-8: Function justification-value

[9] The term negative cycle always refers to a negative cycle in an STP, typically the STP corresponding to the current assignment in the meta-CSP.

the predecessor array [Cormen, Leiserson et al. 1990] when calculating the shortest path array. The predecessor array should also be updated by **propagate** in Figure 4-4. When a value $c : y - x \leq b$ completes a negative cycle (i.e. the FC-condition holds), we follow the predecessor array to retrieve the shortest path p from y to x and return **vars**$(p \cup (x, y))$, where (x, y) is the edge from x to y. The pseudo-code for implementing the predecessor array scheme is given in Figure 4-8. Function **justification-value** returns the justification (i.e. culprit set of variables) for the removal of value $c : y - x \leq b$ from the domain of its variable given an STP S that corresponds to the current assignment. For the efficient implementation of this function, S should be a data structure that stores the predecessor and distance arrays (maintained incrementally by **propagate**). Then function **shortest-path**(y, x, S) can efficiently discover the shortest path from y to x given S. Notice that there may be many negative cycles that justify the removal of some value c, however, both schemes for discovering the justifications will return only one of them.

4.3.2 Removal of Subsumed Variables

The main idea of the heuristic that we will call ***Removal of Subsumed Variables*** is that if a disjunct c_{ij} of a variable C_i is already satisfied by the current assignment A, there is no reason to try other values in the variable's domain (under assignment A): (i) either the current assignment leads to a solution, and since c_{ij} is already satisfied under A, C_i is satisfied in the solution, or (ii) there are no solutions under A and trying other values for C_i will only restrict A even further, thus with no possibility of discovering a solution. We now proceed by formalizing this idea.

Definition 4-6: ***A value c_{ij} is subsumed by an STP*** S (equivalently by an assignment A that implies S) if and only if the constraint c_{ij} always holds in any exact solution of S. A ***variable C_i is subsumed by an STP*** S if and only if there is a variable c_{ij} in the domain of C_i that is subsumed by S.

Let us denote by $d_{xy}(S)$ the distance between time-points x and y in STP S and by $d_{xy}(A)$ the distance between time-points x and y in the STP induced by assignment A.

Theorem 4-3: A value $c_{ij} : y - x \leq b$ is subsumed by an STP S if and only if $d_{xy}(S) \leq b$ (***Subsumption-Condition***), where $d_{xy}(S)$ is the distance between x and y in S.

Proof: "⇐" Suppose that the Subsumption-Condition holds for value c_{ij}. By definition of the distance $y - x \leq d_{xy}(S)$ holds in all exact STP solutions, so $y - x \leq d_{xy}(S) \leq b$ holds in all exact solutions, and c_{ij} holds in all exact solutions of S. "⇒" Conversely, suppose the Subsumption-Condition does not hold, i.e. $b < d_{xy}(S)$ holds. Then there is some exact solution s of the STP for which $y - x = d_{xy}(S)$ [Dechter, Meiri et al. 1991]. Thus, in s, $b < y - x$, i.e. c_{ij} does not hold in all of S's exact solutions.

Lemma 4-1: Adding more constraints to an STP can only result in monotonic decrease in the distances between nodes.

Proof: Immediate, by the fact that adding a constraint to an STP can only reduce the solution set, and thus the distances have to be smaller or equal to the original STP.

Lemma 4-2: Adding a value $c = \{y - x \leq b\}$ to an STP S when S subsumes c, does not change the distance array of S (i.e. the resulting STP is equivalent to the one before the addition).

Proof: Since c is subsumed, by Theorem 4-3, $d_{xy}(S) \leq d$. Let p be a shortest path in the distance array from x to y, and so its weight is $d_{xy}(S)$. Let us suppose now that the distance array of S changes when c is added. That means that the new constraint c participates in at least one shortest path, let us say on the path from w to z that before the addition had distance $d_{wz}(S)$. From shortest path properties we get:

$$d_{wz}(S) \leq d_{wx}(S) + d_{xy}(S) + d_{yz}(S) \leq d_{wx}(S) + b + d_{yz}(S) \quad (1)$$

After the addition, in the new STP $S' = S \cup c$, the new distance $d_{wz}(S')$ is:

$$d_{wz}(S') = d_{wx}(S') + b + d_{yz}(S') = d_{wx}(S) + b + d_{yz}(S) \quad (2)$$

Notice that $d_{wx}(S') = d_{wx}(S)$ and $d_{yz}(S') = d_{xz}(S)$ because b is already participating in the shortest path from w to z and therefore cannot participate on the shortest paths from w to y or from x to z or a cycle would be present on the path from w to z (i.e. the shortest paths would not be simple).

Since the distance array changed, the new shortest path has to be strictly smaller the one than before c was added (Lemma 4-1), i.e.

$$d_{wz}(S') < d_{wz}(S) \quad (3)$$

(1), (2), and (3) imply $b < b$ which is a contradiction, therefore the distance array will not change.

Theorem 4-4: Let $D=<V, C>$ be a DTP, A an assignment on D (i.e. an STP component), and C_i a subsumed variable by A. Then A is a solution of D if and only if it is a solution of $D'=<V, C - C_i>$.

Proof: Since we assume that C_i is a subsumed variable, then, there must be a value $c : y - x \leq b$ in the domain of C that is subsumed by A. Suppose that A is a solution of D. Then obviously, it is a solution of $D'=<V, C - C_i>$ since D' has one less variable (DTP constraint). Conversely, suppose A is a solution of D'. Since c is subsumed by A, it holds in all of A's exact solutions, and thus C_i which is a disjunction involving c, holds in all of A's exact solutions. Thus, if A is a solution of the DTP constraint $C - C_i$, it also solves the DTP constraints C.

Corollary 4-2: Let A be a partial assignment during a DTP search (e.g. algorithm **Improved-DTP** in Figure 4-4) and U some unassigned variables. If A can be extended to a solution over variables in $U - Sub$, it can be extended to a solution over variables in U, where Sub is the set of subsumed variables in U. In other words, we can remove the subsumed variables from the unassigned variables during search. The solution to the reduced problem is a solution to the original problem.

Proof: By **Theorem 4-4** if A' is an extension of A over the variables at $U - Sub$, and A' is consistent, then A' is also a solution to the original DTP.

$C_1 : \{c_{11} : y - x \leq 5\} \vee \{c_{12} : w - y \leq -10\}$
$C_2 : \{c_{21} : x - z \leq 5\}$
$C_3 : \{c_{31} : y - z \leq 15\} \vee \{c_{32} : z - v \leq 10\}$
$C_4 : \{c_{41} : z - v \leq 5\} \vee \{c_{42} : y - w \leq -10\}$
$C_5 : \{c_{51} : v - y \leq -20\} \vee \{c_{52} : z - x \leq -10\}$
$C_6 : \{c_{61} : z - v \leq 2\} \vee \{c_{62} : x - y \leq -10\}$

Figure 4-9: Example DTP for removal of subsumed variables and semantic branching.

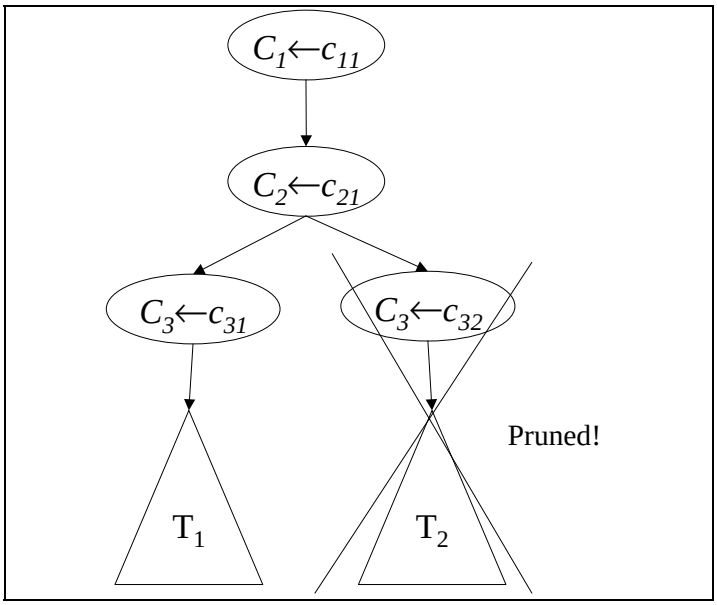


Figure 4-10: The search tree showing the effects of the removal of subsumed variables.

Example 4-3: The ramifications of the above corollary are shown pictorially in Figure 4-10 on the DTP of Figure 4-9. The order the variables are considered is $C_1 - C_6$. In the beginning $A_1 = \{C_1 \leftarrow c_{11}\}$ and immediately afterwards this is extended to $A_2 = \{C_1 \leftarrow c_{11}, C_2 \leftarrow c_{21}\}$. Without removing the subsumed variables, the next assignment would be $A_3 = \{C_1 \leftarrow c_{11}, C_2 \leftarrow c_{21}, C_3 \leftarrow c_{31}\}$. Notice though that value c_{31}, and thus variable C_3 is subsumed by A_2. This means that by Corollary 4-2 C_3 could safely have been removed from the search underneath the subtree of A_2. Without the removal of C_3, when the search of subtree T_1 in figure fails, as it will in this particular example, the search continues by trying the other value of C_3 and so $A_4 = \{C_1 \leftarrow c_{11}, C_2 \leftarrow c_{21}, C_3 \leftarrow c_{32}\}$. Again, by Corollary 4-2, since A_3 has failed, A_4 will fail too. By removing the subsumed variable C_3 in this particular example, subtree T_2 and the node corresponding to A_4 in the figure would have been safely pruned.

4.3.3 Semantic Branching

Semantic branching (***SB***) has been proven an extremely powerful pruning method and was first applied to DTP solving in [Armando, Castellini et al. 1999].

The basic idea of semantic branching is the following: Suppose that during search the assignment $A \cup \{C_i \leftarrow c_{ij}\}$ is expanded but it leads to no solution. That means that in any solution that is an extension of A, if there is any, the value/constraint c_{ij} does not hold. Thus, the negation of this constraint has to hold in any such solution. In other words, if c_{ij} is the constraint $x - y \leq b$ and we know this does not hold, then in any solution that is an extension of assignment A, $\neg c_{ij}$ has to be true, i.e. $y - x < -b$. What this implies is that when search "branches" after failing to extend $A \cup \{C_i \leftarrow c_{ij}\}$ to a solution, and tries a different value for C_i for the rest of the search under A, we can assume $\neg c_{ij}$ holds. The constraint $\neg c_{ij}$ tightens the STP that corresponds to the current assignment A, and thus it is possible that values in other variable domains will be removed earlier than when the constraint was missing.

Notice that with SB the current assignment A and the current STP S (see Figure 4-4) are not in one-to-one correspondence any more. S is union of the values assigned to variables in A *and* all the current semantic branching constraints. We will denote the values of an assignment A as ***values(A)***. In the discussion of the previous paragraph, without SB when $A \cup \{C_i \leftarrow c_{ij}\}$ fails the next assignment is $A \cup \{C_i \leftarrow c_{i(j+1)}\}$ and the next current STP S is *values*$(A \cup \{C_i \leftarrow c_{i(j+1)}\})$. With SB, the next assignment is again $A \cup \{C_i \leftarrow c_{i(j+1)}\}$ but the next current STP S is *values*$(A \cup \{C_i \leftarrow c_{i(j+1)}\}) \cup \neg c_{ij}$.

Example 4-4: We now present an example run of the algorithm at Figure 4-4 to show how semantic branching effectively prunes the search space. First, we will trace the algorithm on the DTP of Figure 4-9 without semantic branching and then *with* semantic branching so as to pinpoint the differences. Suppose the algorithm has already assigned $A_1 = \{C_1 \leftarrow c_{11}\ ,\ C_2 \leftarrow c_{21}\ ,\ C_3 \leftarrow c_{31}\}$ as in Figure 4-11. On the left the search of the meta-CSP is shown and on the right is the implied STP. The x-crossed edges are the ones removed by forward checking while the filled nodes are the ones that belong to the current assignment. In the next node (Figure 4-12) the assignment is extended to $A_2 = \{C_1 \leftarrow c_{11}\ ,\ C_2 \leftarrow c_{21}\ ,\ C_3 \leftarrow c_{31}\ ,\ C_4 \leftarrow c_{41}\}$ which fails because both values in $D(C_5)$ are removed.

The search then continues by trying a different value for C_4 (Figure 4-13). Finally, the search reaches a dead end again because both values in $D(C_6)$ are removed (Figure 4-14), after which it will backtracks back to node C_3 and continue the search.

Had we used semantic branching however, when we branched to try the second value of C_4 we can assume that the constraint $\neg c_{41}$ holds. This is shown at Figure 4-15 (notice the extra constraint $\neg c_{41}$ on the branch). The constraint $\neg c_{41}$ (shown in bold at the right picture) allows forward check to immediately eliminate value c_{61} thus reaching a dead end. In this simple example, SB pruned only one node, the one that assigns $C_5 \leftarrow c_{51}$ (last node in left picture of Figure 4-14), but in general SB can prune an arbitrarily large number of nodes.

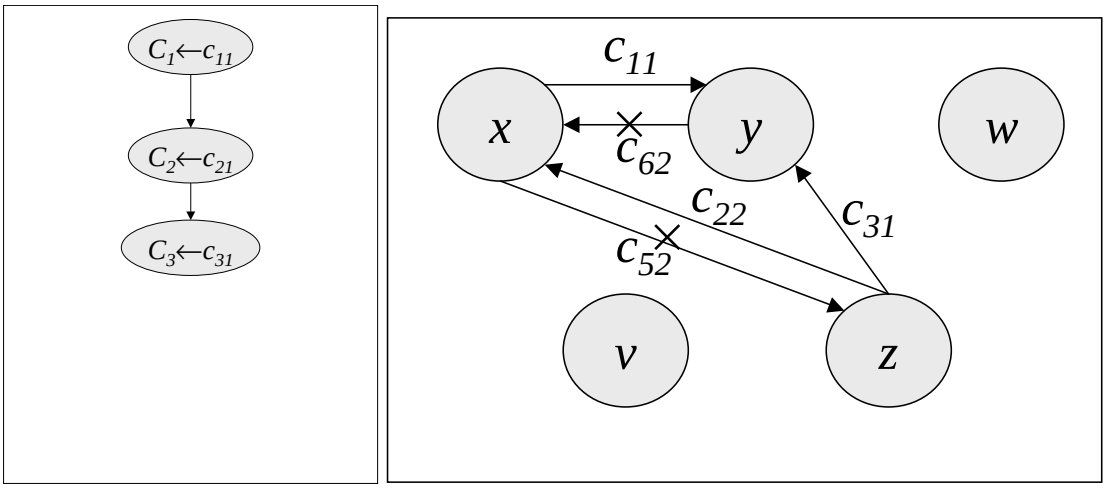


Figure 4-11: Semantic Branching example (a)

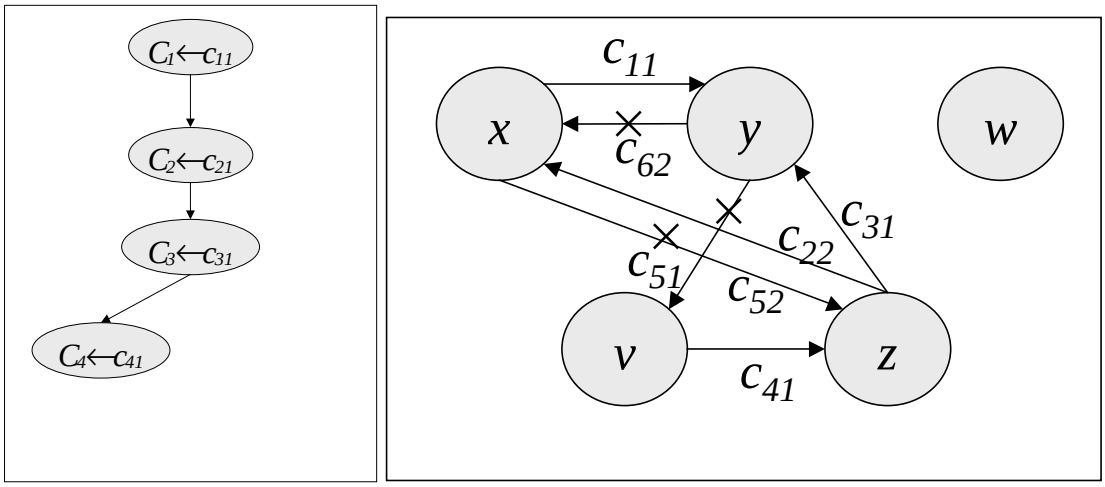


Figure 4-12: Semantic Branching example (b)

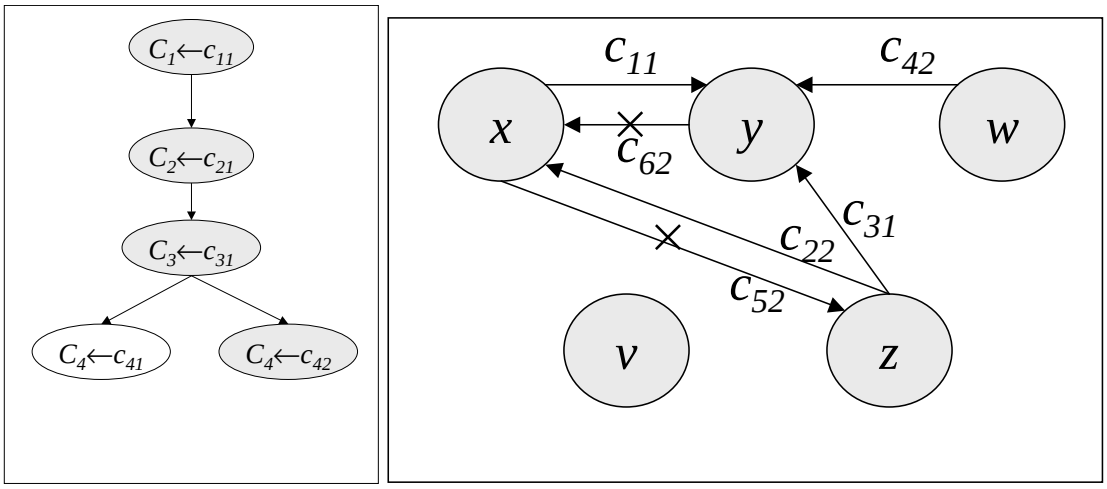


Figure 4-13: Semantic Branching example (c)

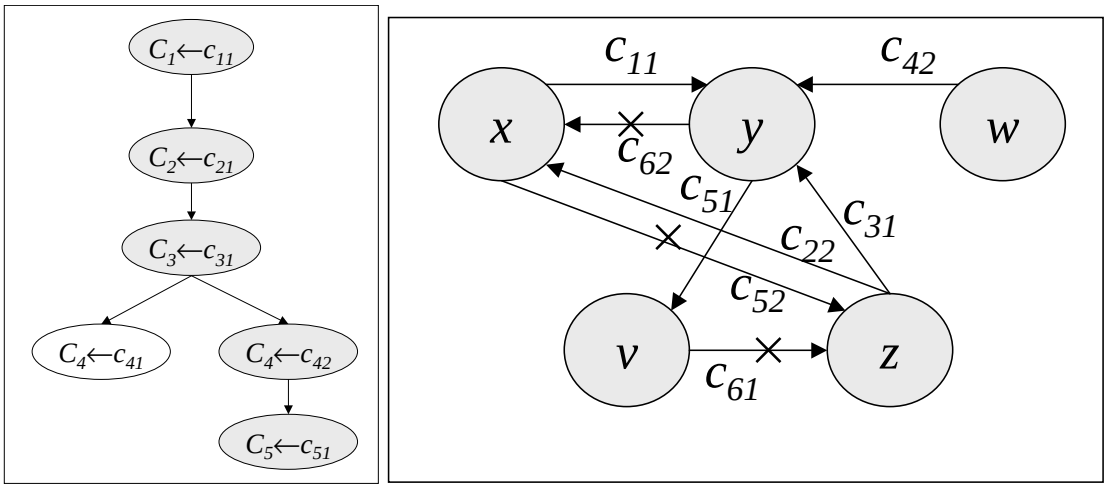


Figure 4-14: Semantic Branching example (d)

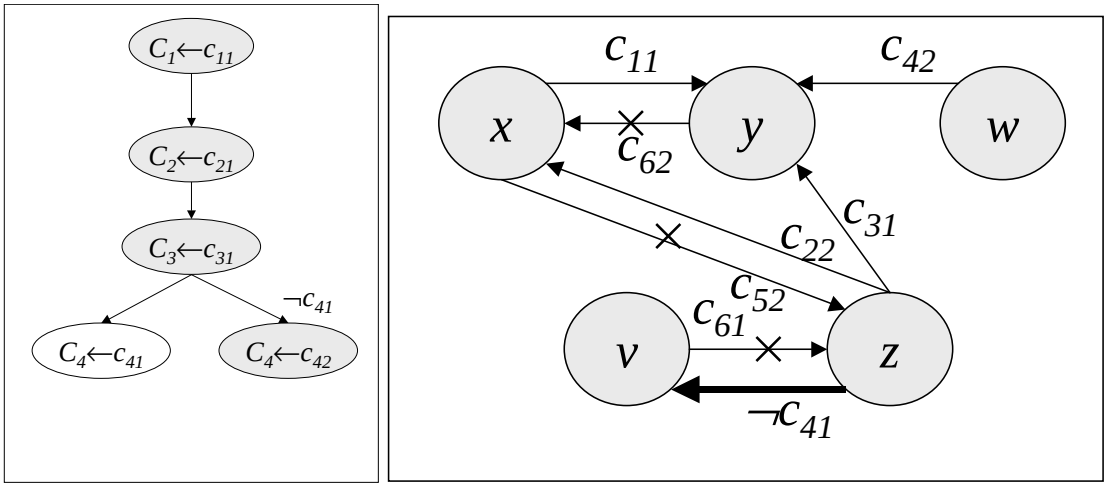


Figure 4-15: Semantic Branching example (e)

4.3.4 Other techniques used in DTP solving: Forward Checking Switch-off and Incremental Forward Checking

Another technique used in DTP in [Stergiou and Koubarakis 1998; Stergiou and Koubarakis 2000] is what we will call ***Forward-Checking Switch-off*** **(FC-Off)**. With FC-off, when the current domain $d(C_i)$ of a variable C_i has been reduced to a singleton set $\{c_{ij}\}$, forward checking is suspended and the constraint c_{ij} is assigned to C_i without forward checking. FC-off is illustrated in the following example:

Example 4-5: Consider the following DTP:

$$C_1 : \{c_{11} : y - x \leq 5\}$$
$$C_2 : \{c_{21} : x - z \leq 5\}$$
$$C_3 : \{c_{31} : v - x \leq 5\} \vee \{c_{32} : z - v \leq 10\}$$
$$C_4 : \{c_{41} : y - z \leq -10\} \vee \{c_{42} : x - y \leq -10\}$$

Without FC-off, when the current assignment becomes $A_1 = \{C_1 \leftarrow c_{11}\}$, forward check will remove variable c_{42} from $d(C_4)$. In the next step, when the current assignment is $A_2 = \{C_1 \leftarrow c_{11}\ ,\ C_2 \leftarrow c_{21}\}$, c_{41} is also removed and thus $d(C_4)$ becomes empty and the search returns failure. In contrast, when FC-off is used, when the current assignment is $A_1 = \{C_1 \leftarrow c_{11}\}$ forward check is suspended and nothing is removed from any variable domain. Similarly, when the current assignment is $A_2 = \{C_1 \leftarrow c_{11}\ ,\ C_2 \leftarrow c_{21}\}$, nothing is removed from any variable domain since forward checking is not performed. Only when the current assignment becomes $A_3 = \{C_1 \leftarrow c_{11}\ ,\ C_2 \leftarrow c_{21}\ ,\ C_3 \leftarrow c_{31}\}$ and the non-singleton domain $d(C_3)$ is encountered is forward check called and able to determine that we have reached a failure. To compare the forward checking DTP solving algorithm that does not use FC-off with the one that does, notice that without FC-off we have forward checked 5+3=8 values and expanded two nodes (i.e. 5 checks for A_1, which checks all other values and removes one of them, and 3 checks for A_2). With FC-off we have forward checked 2 values (we forward check only when the current assignment is A_3) and expanded three nodes.

The above example shows that FC-off may or may not increase the performance of a DTP solving algorithm. With FC-off there is less forward checking but more nodes are expanded (Theorem 17 in [Stergiou and Koubarakis 2000]). Therefore, *performance cannot be analyzed solely in terms of the number of forward checks*: a performance increase depends on the time cost of each forward checking operation, the cost of node expansion, the increase in the number of nodes expanded, and the decrease in the number of forward checked values. There has not been a theoretical analysis that proves that the combination of forward checking with FC-off necessarily improves performance or under which circumstances, but we provide experimentation work in a later section.

Like FC-off, a technique called ***incremental forward checking*** (***IFC***) that appeared in [Armando, Castellini et al. 1999] also aims in reducing the number of constraints checks performed during search, but unlike FC-off this technique does not increase the number of nodes expanded. In IFC only those values that actually need to be forward checked really are. Recall that to forward check a value we just need to check if the FC-condition is true, i.e. for a

value $c_{ij} : y - x \leq b_{xy}$ if $b_{xy} + d_{yx} < 0$, where d_{yx} is the distance between y and x in the current STP. Therefore, if d_{yx} has not changed from the previous time that we have performed forward checking, we do not need to recheck the FC-condition for that value.

IFC initially constructs a $|V| \times |V|$ table named AMV (from the terms affected meta-values according to [Armando, Castellini et al. 1999]), where $|V|$ is the number of DTP variables. Then AMV is initialized so that AMV(x, y) contains a list of pointers to all values c_{ij} of the form $x - y \leq b_{xy}$. When constraint propagation is performed (e.g. by function **maintain-consistency**, line 5, Figure 4-4) the distances that were affected by the propagation are kept in a list D. Subsequently, when forward checking is performed, for each distance d_{xy} in D, all the values in AMV(x, y) are forward checked. Notice that these values are the only ones that could possible have been affected by the propagation of the new constraint.

IFC is a technique orthogonal to all previous pruning methods discussed and could be applied in combination with any of them.

4.4 Integrating all Pruning Methods: The Epilitis Algorithm

In this section we present a new algorithm, called Epilitis, which combines all the pruning methods that were used so far in the literature, namely Conflict Directed Backjumping, Removal of Subsumed Variables, and Semantic Branching, and adds in along with the no-good recording scheme of Figure 3-6 adopted for DTPs, as well as the FC-off technique mentioned above. The main difficulty in designing the algorithm is that no-good recording, CDB and SB interact and thus special attention is required to combine them. The complete algorithm is shown in Figure 4-16.

In Section 4.8.4 we compare Epilitis to the previous state of the art DTP solver, that of ACG, and show an improvement of about two order of magnitude on randomly generated DTP problems.

The algorithm is presented in detailed pseudo-code because it is our belief that any description at a higher level will miss details absolutely necessary for correctly re-implementing the algorithm. This choice increases the complexity of the description but nevertheless has the advantage of removing the guesswork from any researchers that are interested in coding the algorithm. We will now explain the algorithm in detail.

First notice that the Epilitis is a generalization of the no-good recording algorithm of Figure 3-6, i.e. it just adds code to the previous algorithm. The lines common to both algorithms are annotated with an asterisk following the line number. Epilitis would still correctly solve DTP problems if only these lines are included. The only difference from the plain no-good recording algorithm would be in forward checking. Epilitis' forward checking mechanism is similar[10] to the one in Figure 4-4, which takes into consideration the fact that the values of the meta-CSP express STP-like constraints. In other words, take the algorithm of Figure 3-6, change the forward-checking function, and you have a no-good recording DTP solver.

[10] In the no-good recording algorithm the function **forward-check** should return a justification but the forward check function of Figure 4-4 does not. We will present the correct version of forward-check for Epilitis in a subsequent section in full detail.

```
1. Epilitis(A, U, S) /* A is the set of assigned CTP variables, U the set of unassigned variables, and S a distance array representing the current STP */
2*    If U=∅ Then
3*        A is a solution, Stop
4*    Else
5*        Let x be a variable in U, J =∅, BJ = false /* choose x according to a heuristic, set justification J to empty */
6. SB     SBJ =∅          /* and backjumping flag BJ to false. Set the semantic branching justification to empty */
7. RS     If there is value v∈ d(x) subsumed by S Then
8. RS             Return Epilitis(A , U – {x }, S)
9. RS     EndIf
10*       For each v ∈ d(x) until BJ        /* loop for all values in the current domain or until the BJ flag is true */
11*               A' = A ∪ {x ← v }                                   /* add value to a (new) assignment */
12.               S' = maintain-consistency(v, S)
13.               If S' is inconsistent Then
14. SB                    SBJ = justification-value(v, S')
15.                       J = J ∪ SBJ
16.                       GoTo 36
17.               EndIf
18. FC-off        If d(x) is singleton          /* when FC-off omit forward checking when d(x) is singleton */
19. FC-off                K = ∅
20. FC-off        Else
21*                       K be forward-check(A', U, S')
22.               EndIf
23*               If K = ∅                                            /* If we have not reach a dead end … */
24*                       Let J-sons be Epilitis(A', U – {x}, S')     /* then recursively call Epilitis */
25*CDB                    If x ∈ J-sons Then
26*                               J ← J ∪ J-sons
27*CDB                    Else      /* If the current variable does not participate in the failure justification */
28*CDB                            J ← J-sons, BJ=true                 /* then exit the loop */
29*CDB                    Endif
30. SB                    SBJ =J-sons
31*               Else
32*                       J ← J ∪ K
33*NG                     record(project(A', K), K)
34. SB                    SBJ = K
35*               Endif
36. SB            If v is not the last value in d(x)        /* remember SB-constraints is the only global variable */
37. SB                    S=maintain-consistency(S, reverse(v)); SB-Constraints=SB-Constraints∪<reverse(v), SBJ>
38. SB                    If S is inconsistent
39. SB                            un-forward(x), FinishLoop
40. SB                    EndIf
41. SB            EndIf
42*               un-forward(x)
43*       EndFor
44*       If BJ = false Then
45*               For each v ∈ D(v) – d(v) /* add the justifications that removed the values from the current domain */
46*                       J ← J ∪ killers(v)
47*               EndFor
48*NG             record(project(A, J), J)
49*       Endif
50. SB    Remove all semantic branching constraints added in this invocation of Epilitis from SB-Constraints
51*       Return J
52*    Endif
```

Figure 4-16:The Epilitis Algorithm

Having said that, two points must be explained regarding the workings of Epilitis: (i) the additional ("un-starred") lines in the algorithm and (2) the exact way forward checking is performed. The former is the topic of the rest of this section, and the latter the topic of the next one.

For the rest of the discussion let us assume that there is available a function **forward-check**(*A, U, S*) that given the assignment *A*, removes from the variables *U* all the values in their current domains that are inconsistent with *A*. To check the FC-Condition efficiently we provide to the function the distance array *S* that corresponds to *A*. If a domain of a variable is reduced to the empty set **forward-check** should return a justification *K* (also called the *value Killers* of the domain values; value killers are described in more detail in Section 3.5.1), which is a minimal set of variables in *A* such that the constraints among them cause all the variables of the domain to be removed. This description of **forward-check** is exactly the same as the one used so far for the non-temporal CSP solver of Figure 3-6 (with the exception that *S* is not passed as a parameter there).

The annotations **SB, FC-off, CDB, NG** and **RS** next to a line in the algorithm indicate which of the corresponding technique (semantic branching, FC-off, conflict directed backjumping, no-good recording, and removal of subsumed variables respectively) the line serves. For example, to remove semantic branching from the algorithm, we could just remove the lines annotated with **SB**. This way, the algorithm also serves as a test-bed for experimenting with and testing the effectiveness of any combination of pruning techniques. This flexibility is taken advantage of in the experimental section.

The ***Removal of Subsumed Variables*** (RS) in lines 7-9 is achieved by testing if the next variable to assign, there is a value that is subsumed by the current STP *S*. The test can be achieved by checking the Subsumption-Condition of Theorem 4-3. If the variable is subsumed, then it is removed from the unassigned variables and Epilitis is recursively called.

Line 12 propagates the value/constraint of assignment $\{x \leftarrow v\}$ in the current STP *S'* so that the distances of the STP corresponding to the current assignment *A'* are available with a simple table lookup. Recall that the distances are required to calculate both the FC-Condition and the Subsumption-Condition. This technique of maintaining the distance array was described in full in Section 4.2.1 and presented in the algorithm in Figure 4-4.

When only the simple forward checking DTP solving algorithm is used, no assignment $\{x \leftarrow v\}$ will ever cause an inconsistency to the current STP because, if it did it would have been removed by forward checking. However, when semantic branching is used, it becomes necessary to check that the constraint $\{x \leftarrow v\}$ added by semantic branching does not cause an inconsistency. This is the reason for the check at line 13. If we have indeed hit an inconsistency the reason for it is accumulated in variable *J* (line 15), which is the justification to return in case of a failure (line 50). Line 14, annotated with **SB** is explained in the discussion of semantic branching below.

Next the algorithm performs FC-off (lines 18-20). Very simply, if there is only one value in the current domain of the current variable, we omit forward checking and assume that it succeeded by setting $K = \varnothing$. The ramifications of the omission were the subject of Section 4.3.4 above. Otherwise, we perform forward checking and store in *K* the *value killers* of the domain that was reduced to the empty set or we store ∅ to *K* is there is no such domain.

If we have not hit a dead-end, i.e. $K=\emptyset$ (line 23), then we recursively call Epilitis. If it returns, then we have failed to extend the current assignment *A'* to a solution and the return value is a justification of the failure, stored in the variable *J-sons*. If the current variable participates in this justification (line 25) then we accumulate the justifications in variable *J* and proceed (after line 36) trying new values for the current variable. If on the other hand, the current variable has nothing to do with the failure, we jump to line 28, where *BJ* (from backjumping) is set to *true* so that we exit the loop and avoid trying any other values of the current variable, and finally return the same reason *J-sons* that caused the failure in the recursive call. On the other hand, if forward check fails, it returns the value killers *K* that are accumulated in the overall justification *J* (line 32). Line 33 records the no-good implied by the dead-end. Lines 23-35 (apart from the addition of lines 30 and 34) are exactly the same as in the non-temporal no-good recording algorithm.

The most complicated addition is perhaps the lines that achieve semantic branching. Integrating SB with the rest of the pruning techniques has implications that also affect the details of forward checking but for the moment we restrict the discussion only to the code that appears in Figure 4-16. When the current assignment $A \cup \{x \leftarrow v\}$ fails to be extended to a solution, the code reaches line 36. As already described in detail in Section 4.3.3, for the rest of the search under assignment *A*, we can assume that $\neg v$ does not hold. Thus, line 36 propagates the **reverse** of *v* in the current STP *S* (i.e. the STP that corresponds to assignment *A*). The propagation might cause an inconsistency which would be identified at line 38 in which case there is no reason to try a different value for variable *x* and we can exit the loop.

For reasons that will become apparent in the next section, it is necessary to store the semantic branching constraints that we propagate along with the justification for their addition. The store occurs at line 37 where pairs <*v, SBJ*>, where *v* is an STP-like constraint and *SBJ* a justification (from semantic branching justification) are stored in the global variable *SB-Constraints*. There are three different reasons why the current value *v* causes an inconsistency, and correspondingly, three different lines where *SBJ* is assigned a value. Value *v* might directly cause an inconsistency in the current assignment *A* in which case *SBJ* is the justification discovered by function **justification-value** (Figure 4-8)[11] and the assignment takes place at line 14. Alternatively, if after assigning value *v* forward check failed, then *SBJ* should be the value killers *K* that forward check returned (line 34). Finally, if after assigning value *v* forward checked succeeded but the recursive call to Epilitis failed, then *SBJ* is assigned the value *J-sons* (line 30). In all three cases, we fail to extend $A \cup \{x \leftarrow v\}$ to a solution and we store to *SBJ* the reason why not (i.e. the culprit of the variables participating in the constraints the cause the inconsistency).

The rest of the Epilitis algorithm, as already mentioned, is exactly the same as the non-temporal no-good recording algorithm of Figure 3-6.

4.5 Forward Checking and Justifications in Epilitis

Figure 4-16 presented the Epilitis algorithm, however, important details for its implementation were hidden in the **forward-check** and **justification-value** functions. We now proceed to the discussion of these two functions.

[11] Function **justification-value** has to be slightly modified to work for Epilitis as we will see in the next section.

```
forward-check(A, S, U)
1.    For each variable C in U
2.        For each value c : x – y ≤ b_xy in d(C)
3.        If b_xy + distance (y, x, S) < 0 (FC-Condition) or
4.          A ∪ {C ← c } is a superset of A', where <A', no-good-J> is a recorded no-good
5.              Remove c from d(C)
6.              If d(C) = ∅
7.                  K = ∅
8.                  For each value v in D(C)
9.                      K = K ∪ justification-value(v, A, S)
10.                 EndFor
11.                 return K
12.             EndIf
13.       EndIf
14.   EndFor
15.   Return ∅
```

Figure 4-17: Forward Checking for Epilitis

Recall that we assumed that **forward-check**(*A, U, S*) is a function that given the assignment *A*, removes from the variables *U* all the values in their current domains that are inconsistent with *A*. The STP *S* containing the distance array that that corresponds to *A* is passed to efficiently check the FC-Condition. A very important feature of **forward-check** is that if a domain of a variable is reduced to the empty set, it should return a justification *K* (also called the *value killers*), which is a minimal set of variables in *A* whose constraints among them cause the variables of the domain to be removed.

Forward-check should check if each remaining value *v* in some current domain of a variable should be removed or not. A value *v* should be removed, if, as before, the FC-Condition holds, but also if $A \cup \{x \leftarrow v\}$ is a superset of some recorded no-good $<A', J>$, i.e. if $A' \subseteq A \cup \{x \leftarrow v\}$. That achieves forward checking, but it does not solve the problem of assembling and returning a justification in the case where a variable domain is reduced to the empty set. Let us suppose that **justification-value**(*v, S*) is responsible for returning the justification of the removal of a single value *v* given the current STP *S*. Then, the overall justification for a variable domain being empty is the union of the justifications for removing each value originally in that domain. Function **forward-check** for Epilitis is shown in Figure 4-17.

Now we can turn our attention to the implementation of the function **justification-value**(*v, A, S*) which should return a set of variables from *A*, i.e. a justification, that explains why $A \cup \{C \leftarrow v\}$ cannot be extended to a solution and thus *v* has to be removed from the current domain of *C*. Trivially, all the variables in *A* plus the variable *C* constitute a justification for the removal of *v*. However, we can find smaller justifications that provide more opportunities for conflict directed backjumping and search pruning.

There are two reasons why a value *v* might be removed. The first one is if $A \cup \{C \leftarrow v\}$ is a superset of *A'*, where $<A', J>$ is a recorded no-good, and then the justification for removing *v* is

J'. The second reason is if v is removed because $A \cup \{C \leftarrow v\}$ corresponds to an inconsistent STP *S*. Then, as explained in detail in Section 4.3.1 the variables that cause the inconsistency are the ones that have values assigned to them that participate in a negative cycle in *S*. Figure 4-8 for example implements the latter case.

As already mentioned in Section 4.3.3, when semantic branching is present, the current STP *S* does not directly correspond to the current assignment *A*, but instead is formed by all the constraints in *A* plus the semantic branching constraints added. Thus, in Epilitis, when the negative cycles are identified, the corresponding justification is not just the variables with values that participate in the cycle, but also the variables (i.e. the justifications) that were responsible for the addition of the semantic branching constraints that participate in the cycle. These justifications can be found in the *SB-Constraints* structure: whenever a semantic branching constraint is propagated the pair $<\neg v, SBJ>$ is stored at *SB-Constraints* (line 37, Figure 4-16). Function **justification-value** modified for Epilitis is shown in Figure 4-18.

4.6 Implementing the No-good Look-up Mechanism

To make use of the no-goods discovered during search, the algorithm employs them for forward checking. In particular, at line 4 of the **forward-check** algorithm (Figure 4-17), when the value v is being forward checked, the recorded no-goods are searched and if any of them is a

justification-value($c : y - x \leq b$, A, S}
1. If $A \cup \{C \leftarrow c\}$ is a superset of A', where $<A', J'>$ is a recorded no-good
2. Return J'
3. Else
4. p = **shotest-path**(y, x, S)
5. Return **vars**($p \cup c$} $\cup$ {J, where $<v, J> \in$ *SB-Constraints* ***and*** $v \in p$}
6. EndIf

Figure 4-18: The function just-value for Epilitis

subset of the current assignment A plus the value v, i.e. $A \cup \{C \leftarrow v\}$, v should be removed from the domain of *C*. We will call this operation of discovering a no-good $<A', J'>$, such that $A' \subseteq A \cup \{C \leftarrow v\}$, **lookup**($A \cup \{C \leftarrow v\}$). **Lookup** should return J' if there is such a no-good $<A', J'>$ and *false* otherwise. A brute force algorithm for **lookup** is to go through all the recorded no-goods one by one and check if $A' \subseteq A \cup \{C \leftarrow v\}$. Since forward checking a value is one of the fundamental operations in Epilitis the performance of the algorithm greatly depends on the efficiency of **lookup**. The number of no-goods recorded can grow exponentially in the worst case to the size of the DTP and thus the brute force method above, even though it is linear to the number of recorded no-goods, it is not efficient enough. This section presents one scheme for implementing **lookup** that is sub-linear in the number of recorded no-goods. The general idea is borrowed from SAT solvers and the way they check whether a clause has been satisfied.

With our scheme, in order for **lookup** to be efficiently implemented, the no-goods have to be stored and accessed in a particular way. First of all, even though no-goods are pairs $<A, J>$ of an

assignment A and a justification J, they are stored as triplets $<A, J, l>$ where l is the length of the "yet-unsatisfied" part of the assignment A. The exact role of l will become clear in the discussion below. Each value c_{ij} in the DTP has associated with it a no-good list $NG(c_{ij})$ that contains pointers to all no-goods $<A, J, l>$ such that $\{C_i \leftarrow c_{ij}\}$ is part of A. For example, when the no-good $n_1 = <\{C_1 \leftarrow c_{11}, C_2 \leftarrow c_{21}, C_3 \leftarrow c_{32}\}, J, 3>$ is first recorded, a pointer to it is stored in all $NG(c_{11})$, $NG(c_{21})$, $NG(c_{32})$; after a second no-good $n_2 = <\{C_1 \leftarrow c_{11}, C_2 \leftarrow c_{21}, C_4 \leftarrow c_{41}\}, J, 3>$ is also recorded, the no-good lists contain pointers to the following objects: $NG(c_{11}) = \{n_1, n_2\}$, $NG(c_{21}) = \{n_1, n_2\}$, $NG(c_{31}) = \{n_1\}$, $NG(c_{41}) = \{n_2\}$.

When during search an assignment A is extended over a variable C as $A \cup \{C \leftarrow c\}$ then $NG(c)$ is traversed and all no-goods in it have their l counter decreased. This is because all no-goods in $NG(c)$ contain the sub-assignment $\{C \leftarrow c\}$, which has just been satisfied and thus the "yet-unsatisfied" part of the no-good has been reduced. Conversely, when we backtrack from assignment $A \cup \{C \leftarrow c\}$ so to try a different value for C, the l counter is increased in all no-goods in $NG(c)$. Thus, the l counter of a no-good always contains the number of sub-assignments $\{C \leftarrow c\}$ that have to be done in order for the no-good to be satisfied.

Let us suppose now that we are forward checking value v under current assignment A, in other words we would like to know if there is a no-good $<A', J, \rhd>$ such that $A' \subseteq A \cup \{C \leftarrow v\}$. Notice, that during search since we always forward check all values v before we extend an assignment A over some variable, the current assignment can never be a super-set of a no-good. Thus, if there is a no-good $<A', J, \rhd>$ such that $A' \subseteq A \cup \{C \leftarrow v\}$ then $\{C \leftarrow v\}$ belongs in A'. In other words, if there is such a no-good it has to be in $NG(v)$ and *we can limit the search to only look up in* $NG(v)$, a potentially significant reduction compared to a search through the whole set of recorded no-goods.

Looking into $NG(v)$ for a no-good $<A', J, \rhd>$ how do we know when A' is a sub-set of $A \cup \{C \leftarrow v\}$? Recall that the counter l denotes the number of yet-unsatisfied sub-assignments so that if the addition of $\{C \leftarrow v\}$ to A makes $A \cup \{C \leftarrow v\}$ the superset of a no-good then that no-good has to have an l counter of 1. In other words, the algorithm for **lookup**$(A \cup \{C \leftarrow v\})$ is simply to traverse $NG(v)$ and check if a no-good in there has a counter l equal to 1. We know present an example to clarify the above discussion.

Example 4-6: Suppose that we have the following three no-goods recorded: $n_1 = <\{C_1 \leftarrow c_{11}, C_2 \leftarrow c_{21}, C_3 \leftarrow c_{32}\}, J_1, 3>$, $n_2 = <\{C_1 \leftarrow c_{11}, C_2 \leftarrow c_{21}, C_4 \leftarrow c_{41}\}, J_2, 3>$, and $n_3 = <\{C_1 \leftarrow c_{11}, C_2 \leftarrow c_{22}, C_4 \leftarrow c_{41}\}, J_3, 3>$. Thus it must be the case that:

$$NG(c_{11}) = \{n_1, n_2, n_3\}$$
$$NG(c_{21}) = \{n_1, n_2\}$$
$$NG(c_{22}) = \{n_3\}$$
$$NG(c_{31}) = \{n_1\}$$
$$NG(c_{41}) = \{n_2, n_3\}$$

We will denote with $l(n)$ the l counter of no-good n. When the current assignment becomes $A_1 = \{C_1 \leftarrow c_{11}\}$, $NG(c_{11})$ is traversed and the l counters of n_1, n_2, n_3 are decreased to become $l(n_1) = 2$, $l(n_2) = 2$, and $l(n_3) = 2$. Next, if A_1 is extended to $A_2 = \{C_1 \leftarrow c_{11}, C_2 \leftarrow c_{21}\}$, $NG(c_{21})$ is traversed and the new values for the l counters become $l(n_1) = 1$, $l(n_2) = 1$, and $l(n_3) = 2$. Now, if we forward check value c_{31} we will discover that $l(n_1)=1$

and thus correctly infer that $A_2 \cup \{C_3 \leftarrow c_{31}\}$ is a super set of some no-good, in this case no-good n_1. Similarly, value c_{41} can be removed by forward checking it, but not c_{22}.

This no-good lookup scheme has a sub-linear time complexity for the average case. For each value c to forward check, only the no-goods stored in $NG(c)$, i.e. the no-goods that contain the sub-assignment $\{C \leftarrow c\}$ have to be checked. In the worst case of course, value c might participate in all recorded no-goods in which case the method will take the same time as the brute force linear time algorithm. In addition, with our scheme it takes only constant time to check whether the no-good assignment is a subset of the current assignment plus the value to be forward checked. This is because all that is required is testing whether the l counter of the no-good is equal to 1 or not. However, our scheme imposes some additional overhead during search since every time the current assignment is extended over a new variable, the l counters for the no-goods in the no-good list of the newly assigned value have to be decreased, and similarly when we backtrack from the current assignment the counters have to be increased. This overhead however is again small since only a limited number of no-goods have to have their counters updated. In practice, the scheme showed a significant improvement over the linear algorithm (i.e. the one that goes through all the no-goods), so much that the latter was quickly abandoned with no further consideration.

It is of course possible that other better and more efficient no-good look up schemes will be designed and we make no claims that this is the most efficient. Even so, the experimental results shown in a later section clearly show that the performance gains from using no-goods clearly outweighs the overhead of identifying, recording, and lookup no-goods during search.

4.7 Other DTP Solvers

In this section we compare and categorize all the previous approaches to DTP solving. There are two previous DTP-solvers that treat component-STP selection as CSP problems and perform a search in the same meta-CSP as Epilitis: that of Stergiou and Koubarakis [Stergiou and Koubarakis 1998; Stergiou and Koubarakis 2000] (hereafter **SK**) and of Oddi and Cesta [Oddi and Cesta 2000] (hereafter **OC**). We can compare these approaches in terms of the implementation of (1) **maintain-consistency**, (2) **forward-check**, (3) the variable ordering heuristic, (4) the value ordering heuristic, (5) and the techniques employs for the search and particularly for pruning the search space. An additional approach, which casts DTP-solving in terms of SAT, is discussed in Section 4.7.3.

4.7.1 The Stergiou and Koubarakis approach

1. Function **maintain-consistency** uses the Incremental Directed Path Consistency (IDPC) algorithm of [Chleq 1995]. Its complexity is $O(|V|^2)$ in the worst-case, where $|V|$ is the number of time-points in the DTP, but it has a very good average case behavior. Using the IDPC means that the current STP is maintained directed path-consistent; among other issues, this design choice affects the way forward check is performed.
2. Function **forward-check** is implemented by adding a value to the current STP S and propagating using again the IDPC algorithm, identifing the inconsistency if there is one, and then retracting the constraint using again IDPC so as to be ready to forward check

the next value. This requires two calls to IDPC with $O(|V|^3)$ in the worst-case for each value to be forward checked. There is of course a much faster algorithm: checking if the FC-Condition holds. The FC-condition requires the distance between two time points, which given that the SK approach maintains the current STP in directed path consistency form, can be found in $O(|V|)$ time in the worst case with the algorithm described at [Chleq 1995]. Even though the implementation of SK did not use this improved forward checking technique, for the theoretical comparison, we will use our improved forward checking method to compare the best potential designs of these approaches.

3. The variable ordering heuristic in SK is the *Minimum Remaining Values* (***MRV***) in which the variable with the fewest remaining values in its domain is selected first. Ties among variables are broken by choosing the variable that contains the time-points that appear the most in the rest of the variables. Each variable may contain many disjuncts and each disjunct contains two time-points, so we can choose the variable that contains the time-point with the maximum appearance in other variables/disjunctions or the variable with the largest sum of appearances of its time-points in other variables/disjunctions. The SK paper does not discusses exactly how the selection is performed.
4. The value ordering heuristic is the same as the tiebreaker heuristic above. The disjunct whose time-points appear in the most in other variables is selected first. SK experimented with the anti-heuristic but with disappointing results.
5. The pruning methods that SK use are forward checking, conflict-directed backjumping (CDB), and unit-propagation (FC-off).

4.7.2 The Oddi and Cesta approach

1. Function **maintain-consistency** uses an incremental full path consistency algorithm (e.g. PC-2) (IFPC). In other words, OC choose to maintain the current STP's distance array (which is equivalent to full path consistency). IFCP also takes $O(|V|^3)$ in the worst-case, where $|V|$ is the number of time-points in the DTP, but its average performance is probably worst than IDPC even though I do not know of any empirical results that support this claim.
2. Function **forward-check** checks the FC-condition and since the distance array is maintain it takes only constant time to forward check a specific value.
3. The variable ordering heuristic is the MRV with no other tie-breaking heuristics.
4. There is no particular value ordering heuristic.
5. The pruning methods that OC use are incremental forward checking, removal of subsumed variables, and semantic branching.

4.7.3 The Armando, Castellini, and Giunchiglia approach

The Armando, Castellini, and Giunchiglia (**ACG**) approach differs from the previous ones in that it treats the component-STP selection not as a meta-CSP problem, but as a SAT problem instead. The exact differences and implications of this will be discussed in Section 4.7.5. However, we can use the above classification scheme to compare the approach:

1. The ACG algorithm does not use **maintain-consistency**. Every time the algorithm requires checking the consistency of a set of STP-like constraints they use a version of the Simplex algorithm for linear programming. Simplex has exponential worst-case performance.
2. Function **forward-check** also uses the Simplex algorithm to determine whether the STP corresponding to the value to be forward checked plus the constraints that correspond to the current assignment are consistent.
3. During a preprocessing step, ACG enhances the SAT formula with clauses (equivalently variables in a CSP-based approach) that correspond to inconsistent pairs of literals (equivalently values in a CSP-based approach). This provides additional guidance to an MRV-like criterion for variable selection: they choose the clause that contains the literal with the greatest number of occurrences in the clauses of minimal-length (which implies they will choose the clause with the literal that participates in the most pairwise inconsistencies with other literals). We will call this heuristic ***Max-Inc*** because essentially it picks the clause with the literal that participates in most pairwise inconsistencies.
4. For variable ordering, they choose the literal that maximizes the number of pairwise inconsistencies in the previous step. Notice here however, that a SAT-based procedure can either choose to branch on the literal c_{ij} or the literal $\neg c_{ij}$. Instead, a CSP-based approach can only branch on c_{ij} , i.e. assign a value to a variable, and never the negation of the value to a variable. From the ACG paper it is unclear which branch (i.e. the positive or the negative) is taken first and how this choice is made.
5. The pruning methods ACG use are forward checking, semantic branching, and unit-propagation.

The table below summarizes the above discussion and comparison between the different approaches. The DPT solving approaches are ordered chronologically according to the date of appearance in the literature.

Technique	SK	ACG	OC	Epilitis
Temporal Reasoning				
Constraint Propagation	Maintain IDPC, $O(\|V\|^2)$	N/A	Maintain IFPC, $O(\|V\|^2)$	Maintain IFPC, $O(\|V\|^2)$
Forward-Checking (time complexity is for each value) [12]	Find distance, check FC-Condition, $O(\|V\|)$	Run an STP consistency checking algorithm, $O(\|V\|^3)$	Lookup distance, check FC-Condition, O(1)	Lookup distance, check FC-Condition, O(1)
Value Subsumption (time complexity is for each value) [13]	Find distance, check Subsumption-Condition, $O(\|V\|)$	Run an STP consistency algorithm[6]	Lookup distance, check Subsumption-Condition, O(1)	Lookup distance, check Subsumption-Condition, O(1)
Searching Methods				
CDB	Yes	No	No	Yes
RS	No	No	Yes	Yes
SB	No	Yes	Yes	Yes
FC-off	Yes	Yes	No	Yes
NG	No	No	No	Yes
IFC	No	No	Yes	No
Heuristics				
Variable	MRV	Max-Inc	MRV	See Section 4.8
Value	The value with the time-points with the most appearances in other values	The value with the most pairwise inconsistencies (but it might be negated)	No specific heuristic	See Section 4.8

Table 4-2: Comparison of all DTP solvers

[12] The time complexity shown is for implementing forward checking with the best known algorithm: Given that in the SK approach STP , the current STP is in directional path consistency, we can find the distance between a pair of nodes in time $O(|V|)$ and check the FC-Condition. As already mentioned, in the actual implementation, SK used a less efficient scheme that takes quadratic time. In the ACG approach, we have to run an STP consistency checking algorithm, while in the actual implementation ACG use Simplex.

[13] Neither SK nor ACG use RS, so these two cells do not refer to their actual implementation. Instead, this is the most efficient way they could have implemented RS had they desired to, given the way they perform temporal propagation.

4.7.4 Time complexity of the various techniques

This section analyzes the time complexity of all the above techniques. The theoretical analysis is not enough to prove which methods or combination of methods are the best, but it does provide some intuition to this extend, and clearly defines the trade-offs that each method favors.

- **Temporal propagation**

  The way the current STP is maintained affects the way forward checking is performed and the subsumption of values is determined (see Section 4.2.1). There are currently three different choices in all of the four DPT solvers. ACG does not propagate the constraints in the current STP and conceptually, the current STP is stored as a directed, weighted graph. In order to check the FC-Condition or the Subsumption-Condition, an algorithm that uses this approach has to run an STP consistency sub-routine from scratch. In the worst case and in every node, such an algorithm takes time $O(|V|^3)$ and thus to forward check all values in a search node, it takes $O((f+s)|V|^3)$, where f is the number of values to be forward-checked and s the number of values to be checked for subsumption. In the worst case s will be the maximum domain size ***mds*** .

  On the other hand, SK maintains the STP in directed path consistency state so it takes only $O(|V|)$ to forward check or check the subsumption of a value. Thus, in every node, the overall time required is $O((f+s)|V| + |V|^2)$, because it takes in the worst case $O(|V|^2)$ to run the incremental directional path consistency algorithm. Finally, OC and Epilitis maintain the distance array (i.e. full path consistency) of the current STP and the necessary checks on every node, take time $O((f+s) + |V|^2)$. This is obviously a better worst-case time complexity, but since on average IDPC is faster than IFPC it is not clear which approach is better.

- **CDB**

  Conflict-Directed Backjumping (without no-good learning) requires time $O(mds \times d)$, where d is the depth of the current node. This is because at depth d a negative cycle can have at most d edges. Once such a negative cycle is traversed for each value (in Epilitis this will be performed at function **justification-value** and **forward-check**), the variables-value sub-assignment that correspond to these edges are found and we can jump back to the most recent variable that participates in the failure. In the experimental section we show empirically that the pruning we get from this method far exceeds the slight overhead it consumes.

- **SB**

  Semantic branching has additional overhead $O((mds-1)P)$, where P is the worst-case time to propagate a value in the current STP (i.e. with IDPC and IFPC this is $O(|V|^2)$. Empirically however, ACG have shown that the additional overhead is small compared to the benefits and SB is a very useful pruning technique. This result was also demonstrated in our experiments.

- **RS**

  Removing the subsumed variables takes time $O(mds)$ since it takes constant time to check the Subsumption-Condition, when the distance array (IFCP) of the current STP is maintained. In other approaches, RS might take longer time as it is discussed in the

Temporal Propagation bullet above. Again the benefits from RS are proven empirically to be greater than the small overhead.

- **FC-off**

  To perform FC-off, we just have to determine whether the current domain of the selected variable is singleton or not, in which case we do not perform forward check. The check to perform FC-off thus is constant. However, the implications of using FC-off to the performance of the overall search are not clear as it have been discussed in Section 4.3.4 and our experiments show that in some cases it is better not to perform FC-off.

- **NG**

  For a full discussion on the complexity and implementation of no-good learning in Epilitis see Section 4.6.

- **IFC**

  Incremental forward checking has a O($mds\ |V|$) preprocessing cost since it has to insert each value to an AMV list (actually in the OC implementation the AMVs are also sorted and the sorting is used to further decrease the number of required forward checks, so the preprocessing step takes longer than O($mds\ |V|$)). The extra computational cost on each node for traversing the AMVs increases only by a constant factor while potentially reduces the number of required forward checks.

- **Variable ordering heuristics**

  The simple MRV heuristic takes time O(v) in every node where v is the number of values still left in the domains of the variables, since it has to count all of them to decide which variable to choose next. Other more complicated heuristics, such as the one used in ACG, take time O(v^2) since for each value we have to count the number of values that it might be pairwise inconsistent with. Whether the benefits from a heuristic are more than the overhead it incurs can only be determined empirically.

- **Value ordering heuristics**

  The heuristics in ACG and SK take time O($mds \times n$), where again mds is the maximum domain size of the variables and v the number of values left in the domains of the variables. This is because for each value in the current domain, all the values left in the other domains have to be checked for pairwise inconsistencies (ACG), or for time-points counts (SK), etc. OC does not use such a heuristic.

We also have to note that even though some of these techniques have been proven to be beneficial this fact has to be taken with caution. Experimental work has proven for example that removing the subsumed variables (RS) is beneficial, but this might not be true when RS is used in combination with other pruning methods. For example, another pruning method might be so powerful that the additional benefits from RS might be negligible and thus the additional overhead of RS might actually slow performance. In addition, the experimental work reported so far in the literature deals with randomly generated DTPs and thus it is uncertain if the observation still holds

in DTPs that appear in real domains[14]. Nevertheless, all the above techniques except NG have really low overhead per search node and we expect them to be beneficial in most circumstances.

4.7.5 SAT versus CSP approach

As already mentioned, the Armando, Castellini, and Giunchiglia (ACG) approach treats the selection of disjuncts to form a consistent STP-component as a SAT problem. By definition, a DTP is a set of constraints:

$$(c_{11} \vee \ldots \vee c_{1n_1}) \wedge$$
$$(c_{21} \vee \ldots \vee c_{2n_2}) \wedge$$
$$\vdots$$
$$(c_{m1} \vee \ldots \vee c_{mn_k})$$

which, if each c_{ij} is taken to be a propositional value, forms a SAT problem. Of course, the difference with non-temporal SAT problems is that in this case the propositions c_{ij} have underlying temporal semantics whose consistency has to be checked.

This approach is different from the CSP-based one taken by the rest of the CSP solvers, including Epilitis. In a CSP-based approach, the meta-variables correspond to constraints and the values to disjuncts. Thus, each search node extends the current assignment by a variable-value sub-assignment $\{C_i \leftarrow c_{ij}\}$. In other words, the CSP-based approaches are branching on the values c_{ij}. In contrast, the SAT approach extends the current assignment by assigning a truth value to a proposition, i.e. $\{c_{ij} \leftarrow True\}$ or $\{c_{ij} \leftarrow False\}$. In other words, the SAT-based approach branches on the truth value of c_{ij}.

We will now examine this difference with a specific example. Let us suppose that we try to solve the following DTP:

$$(a \vee b)$$
$$(c \vee d)$$
$$(e \vee f)$$

The exact semantics of the constraints *a, b, c, e* and *f* are not important for the purposes of this example, but let us suppose that the pairs (*a, c*) and (*a, e*) are propositions inconsistent with each other. Since, the ACG algorithm adds the pairs of inconsistent propositions as clauses, the DTP/SAT problem will be augmented with two more clauses as follows and become:

$$(a \vee b)$$
$$(c \vee d)$$
$$(e \vee f)$$
$$(\neg a \vee \neg c)$$
$$(\neg a \vee \neg e)$$

[14] For example, as it is noted in [Oddi and Cesta 2000], Semantic Branching is only useful, when the disjuncts in each constraint are not self-exclusive. For example, in scheduling applications where the constraints are typically of the form $\{A < B$ or $B < A\}$, when the first disjunct fails, SB will add its negation A > B, having no pruning effect, since the next disjunct $B < A$ is the same constraint.

Assuming that ACG will pick *a* as the first proposition to assign a truth value the search performed is shown in Figure 4-19. ACG starts by assigning *True* to *a*. By propagating the assignment it notices that the clauses $(\neg a \vee \neg c)$ $(\neg a \vee \neg e)$ simplify to $(\neg c)$ and $(\neg e)$. Since unit clauses are preferred over all other clauses, they will be picked up by ACG and so the next assignment will be to assign *False* to both *c* and *e*. By propagating all these truth assignments we are left with the clauses (*d*) and (*f*) and so ACG continues by assigning *True* to *d* and *f*. Assuming that at this point the search fails (e.g. that can happen because of an inconsistency in the current STP) ,

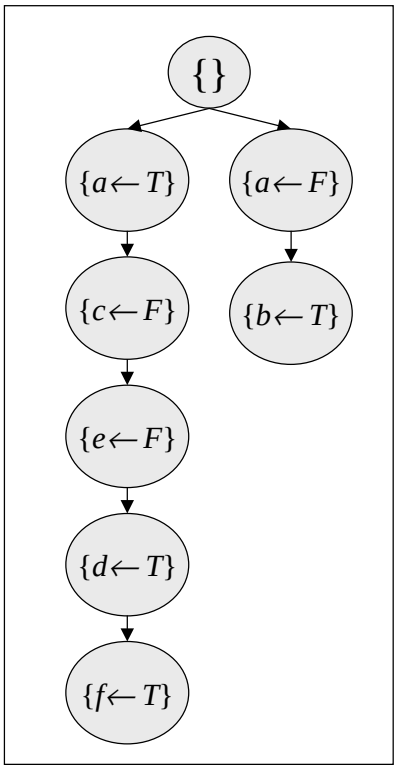


Figure 4-19: The search performed by ACG

ACG will backtrack and continue by trying to assign *False* to *a*. In doing so, the first clause becomes (*b*) and hence this is the next proposition to get a truth-value assigned.

Let us compare the search of Epilitis on the same problem, which is shown in Figure 4-20. First, Epilitis chooses to satisfy the first constraint (correspondingly the first SAT clause) and assigns $\{C_1 \leftarrow a\}$. Forward-checking will remove values *c* and *e* from clauses C_2 and C_3 since, as we assumed, they are pair-wise inconsistent with value *a*. Thus, since the only value remaining in the domain of C_2 is *d* , Epilitis proceeds by assigning $\{C_2 \leftarrow d\}$ and subsequently $\{C_3 \leftarrow f\}$. Again, if this fails, the next assignment will be $\{C_2 \leftarrow b\}$. However, if we also assume Epilitis is using semantic branching, then when we branch on this last assignment, we also propagate the constraint $\neg a$ in the current STP.

So what are the differences between the two approaches? First of all, it is easy to note that ACG expands many more nodes (assuming both approaches use the same exact heuristics). Is this

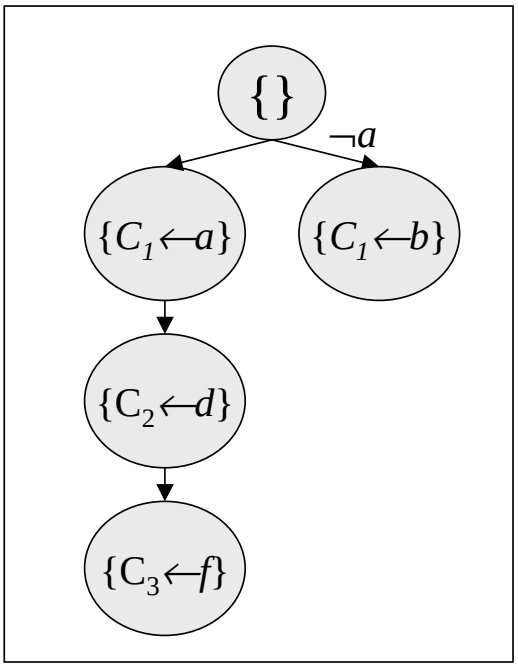


Figure 4-20: The search performed by Epilitis

better or worse? It is neither and here is why. Let us suppose that a is the constraint $x - y \leq d_1$. Since we assumed c is inconsistent with a then it has to be of the form $y - x \leq d_2$, and in addition, $d_1 + d_2 < 0$ (so they form a negative cycle) which implies that $d_1 < -d_2$. The negation of c is thus $x - y < -d_2$, but since $d_1 < -d_2$, the constraint $x - y \leq d_1$ subsumes $x - y < -d_2$. In other words, a subsumes $\neg c$. Thus, the fact that after the assignment $\{a \leftarrow T\}$ ACG proceeds by assigning $\{c \leftarrow F\}$ does not impose any additional constraint to the current STP. It also does not slow down the algorithm since there is no need for the constraint $\neg c$ to be propagated: it is already subsumed by a . In other words the extra nodes in ACG are neither an advantage or disadvantage of the approach[15].

Another difference between the approaches is that semfantic branching comes "naturally" for a SAT-based approach: in our example, when $\{a \leftarrow T\}$ failed we proceed with $\{a \leftarrow F\}$ and then $\{b \leftarrow T\}$. Instead, when the branch below $\{C_1 \leftarrow a\}$ fails, Epilitis continues with $\{C_1 \leftarrow b\}$ and has to add the constraint $\neg a$ to achieve the semantic branching. The end result is the same, however the SAT approach is conceptually simpler and easier to implement. In Epilitis there is the need for semantic branching constraints to be explicitly managed so they can be used to find the justification of failures, while in a SAT-based approach these constraints are already stored and managed as the current assignment.

But, computationally this last observation still does not give the edge to the SAT-based approach. A far subtler difference however does. Notice that in Figure 4-19, had ACG desired so,

[15] The discussion hints that overall performance mostly depends on the number of constraint propagations and not on the actual number of nodes generated in the search. That is, an algorithm might generate more nodes and still have the same or better performance than another algorithm.

it could have begun the search by assigning $\{a \leftarrow F\}$ instead of $\{a \leftarrow T\}$. Similarly, at the last node, it could have assigned $\{b \leftarrow F\}$. In general, a SAT-based approach can potentially first explore the "negative" branch. Epilitis, and all CSP-based approaches do not have this choice: they are forced to assign a value (and not its negation) to a variable C_i.

It is possible of course that there is a relatively easy way to change CSP approaches so that they become as flexible as the SAT approach.

4.8 Experimental Results

In this section we display the results from a series of experiments that we ran on Epilitis and the solver of ACG, publicly available at http://www.mrg.dist.unige.it/~drwho/Tsat. As is customary in the DTP literature [Stergiou and Koubarakis 1998; Armando, Castellini et al. 1999; Oddi and Cesta 2000; Stergiou and Koubarakis 2000], experimental sets are produced using the random DTP generator implemented by Stergiou, in which DTPs are instantiated according to the parameters $<k, N, m, L>$, where k is the number of disjuncts per constraint, N the number of DTP variables, m the number of DTP constraints, and L a positive integer such that for all the disjuncts $x - y \leq b$, $b \in [-L, L]$. In the random DTP problems we used, we used the typical settings in the literature where $k = 2$, $L = 100$, and $N \in \{10, 15, 20, 25, 30\}$. Parameter k is chosen to be 2 because this is the case for constraints that typically appear in many planning and scheduling (e.g. A<B or B<A). Defining R to be the ratio of constraints over variables, m/N , we generated 50 experiments per setting for N, per value of R varying R from 2 to 14 (e.g. when N is 30, R=10, we generated 50 problems with 30 variables and 300 constraints). Thus, the total number of experiment problems were $50 \times 13 \times 5 = 3250$ (13 values for R, 5 different values for N). The domains of the variables are integers instead of reals so that semantic branching can easily be implemented: the negation of the constraint $x - y \leq b$ is $y - x \leq -b - 1$.

The output of Epilitis provides the following statistics for each DTP solved:

- The ***Time*** it took to solve the problem.
- The constraint checks ***CCs*** (forward-checks, i.e. the number times the algorithm checked the FC-condition or the Subsumption-Condition).
- The number of search ***Nodes*** generated.
- The constraint propagations ***CProps*** (i.e. number of calls to **maintain-consistency**).
- The no-goods checks ***NCs***, i.e. the number of times a no-good counter was checked whether it is equal to 1 (see Section 4.6).
- The number of no-goods recorded ***NGs***.

In the graphs and tables showing the results below, we typically display the median of the above statistics over the series of the 50 experiments with the same parameters N and R. Again, this is consistent with the literature on DTP solving. *All the graphs and tables are shown at the end of this section.*

The Epilitis algorithm was implemented in Allegro Common Lisp 5.0. Both Epilitis and the ACG solver were ran on the same Intel Pentium III machine running Windows 2000, having 384MB memory and a clock speed of 1GHz. There is no time-out for the experiments ran by Epilitis while there is a 1000 second time-out on the ACG solver. We tried to set the ACG time-out higher than 1000 seconds but we ran into technical problems: the solver kept crashing.

All of our experiments confirm the existence of a critical region for values *R=5, 6, 7, 8*, where the percentage of solvable problems is less than 10% and the median time to find a solution or prove there is no solution to a DTP problem exceeds the time taken when R<5 or R>8.

4.8.1 Pruning power of techniques

In the first set of experiments we investigated the pruning power of all the pruning methods that we have implemented and combinations of methods. These are:

- Removal of subsumed variables (***RS***)
- Conflict-Directed Backjumping (***CDB***)
- Semantic Branching (***SB***)
- No-good learning (***NG***)

In Epilitis all of the above methods can be individually turned on and off, providing us with the opportunity to try any combination we desire. The only limitation is that whenever *NG* is on, *CDB* must also be turned on. We name our graphs and tables using the following convention: we list the options that were turned on separated by spaces or dashes. When we bound the size of the no-goods we follow the name with the numerical bound. For example, CDB-SB-RS-NG-10 is Epilitis with CDB, SB, RS, NG on and a maximum size of no-goods set to 10, and CDB-SB-RS-NG the same algorithm with no bound on the size of no-goods.

For this set of experiments we used the following ***dynamic*** variable and value heuristics (***H0***):

- Select the variable according to the MRV (*Minimum Remaining Values*). Break the ties by selecting the variable that contains the value that *maximizes* the number of pairwise inconsistencies with the values in the domains of the unassigned variables.
- Select the value that *minimizes* the number of pairwise inconsistencies with the values in the domains of the unassigned variables.

This heuristic is typical in the CSP literature. The idea is that by choosing the variable with the value that maximizes the pairwise inconsistencies, the branching factor of the search is reduced, since this is the variable that most constrains the search. On the other hand, when we select a value we want the value that least constrains the search so that we increase the probability of finding a solution that contains this value.

In Figure 4-21 and Figure 4-22 we show the graphs for the sets of experiments when only one pruning method at a time is on (except as we have already mentioned, whenever NG is on, CDB is on too). All times are reported in seconds, as in all experiments in this dissertation. The "nothing" line corresponds to the Epilitis algorithm with all pruning method turned off and the NG-10 line to the no-good learning algorithm where the size of the no-goods recorded was limited to 10 (i.e. for a no-good $<A, J> A$ could only contain 10 pairs of variable-value assignments). We selected size 10 because in other experiments described in the next sections, size 10 was the optimal size for the Epilitis with all pruning methods on (see Section 4.8.2). As expected "nothing" performs the worst, then RS, then CDB, then SB, NG, and NG-10 following closer together. Notice the scale of the *y*-axis is logarithmic. It makes sense to compare the time of each algorithm, since their underlying implementation is the same.

The experiments for various combinations of pruning methods are in Figure 4-24. As we can see the worst combination is the CDB-RS and the best is the CDB-SB-RS-NG-10. When we did

not bound the size of the no-goods, the performance of the algorithm was seriously degraded for N=30 and this is why the graph for CDB-RS-SB-NG stops at *R*=5.

Table 4-3 combines the results shown so far for all algorithms for N=20. The table shows the median time and orders the columns according to the performance on the center of the critical region where R=6. Thus, the worst method is Nothing and the best is the combination of CDB-RS-SB-NG-10 (i.e. the no-goods had a size of at most 10). SB is the single best pruning method, even better than the combination of CDB-RS. The only N for which we have complete results for all the algorithms is N=20.

Table 4-4 shows the same table for N=25. The results for "nothing" are missing since the algorithm with no pruning method on required too much time to complete. The ordering is almost the same: NG is now better than SB, CDB-SB is better than SB-RS, and CDB-RS-SB-NG is better than CDB-SB-RS-NG-10 (i.e. three pairs of algorithms switched positions).

Table 4-5 shows the same table for N=30. Notice there are no results for any algorithm for which NG is on with no bound on the size of no-goods, because an excessive number of no-goods was generated and recorded, seriously degrading the performance (e.g. the median time of CDB-RS-SB-NG for N=30, R=6 and on the 44 problems that we allowed the algorithm to run before we stop it was 1250 seconds and the average was 13800 seconds). However, when the size of the no-goods was bounded at 10, having the NG option increased the performance resulting in the best algorithm CDB-SB-RS-NG-10 with a median time of 79.8 for R=6, a speedup of about two over the next best algorithm SB-RS[16].

Table 4-6 shows the statistic $Nodes_c/Nodes_{Nothing}$ where $Nodes_c$ are the search nodes explored by the algorithm in column *c* and $Nodes_{Nothing}$ the search nodes explored by Epilitis with no pruning methods. Thus, the table shows the pruning power of each method, ignoring the time overhead to implement the method. As we expect, the more methods we add, the more we prune the search space. In this case NG is the best single[17] method exploring only 27.31% of the whole search space (i.e. when no pruning method is on) for R=6. NG is even better than the combination of all three other methods: CDB-SB-RS. This result encourages us to look for even more efficient implementations of the **look-up** operation for no-goods and reduce the overhead of looking-up no-goods.

Table 4-7 supports the same argument showing the cumulative effect of pruning methods on the search space explored. It displays the results for N=30 where the search space is significantly larger than for N=20 (the search space grows exponentially with N). Each cell $Cell_{RC}$ displays the ratio $Nodes_{R(C-1)} / Nodes_{RC}$, where $Node_{RC}$ is the Median Nodes explored by the algorithm in the *C* column for variables N=30 and ratio *R* (for the first column where *C-1* is undefined, we just show $Nodes_{RC} / Nodes_{RC} = 1$). For example, in the last column, for R=6, we see that this ratio is 41.07% meaning that the algorithm CDB-RS-SB-NG-10 explored only 41.07 of the space the algorithm in the previous column CDB-RS-SB explored on problems for N=30 and R=6. The result shows in an impressive way the pruning power of no-goods. Calculating the overall pruning relative to just SB for N=30, R=6 we find $94.93 \times 41.07 = 39\%$, while for R=5 is $97.13\% \times 95.78\% \times 50.66\% = 47.1\%$.

---

[16] Surprisingly, SB-RS and CDB-SB are better than CDB-SB-RS. However, their difference is very close.

[17] In a way, since when NG is on, CDB is on too.

As we have already mentioned and as is noted in [Oddi and Cesta 2000], Semantic Branching is only useful when the disjuncts in each constraint are not self-exclusive. For example, in scheduling applications where the constraints are typically of the form $\{A < B$ or $B < A\}$, when the first disjunct fails, SB will add its negation A > B, having no pruning effect, since the next disjunct $B < A$ is the same constraint. However, the no-good recording algorithm should still be effective. Thus, even though SB is a very powerful pruning method, NG is effective in more cases. The effectiveness of NG in real DTPs or DTPs with certain structure is to be determined.

Summarizing the results of this section:

- A rough partial ordering of the pruning methods is RS < CDB < CDB-RS < NG-10 < SB < {CDB-SB, SB-RS, CDB-SB-RS} < CDB-SB-RS-NG-10.
- No-good learning needs to bound the size of the no-goods recorded because asymptotically the overhead of recording and looking-up the no-goods greatly outweighs the benefits.
- SB is the best single pruning method in terms of performance, i.e. displaying a good trade-off between pruning power and implementation overhead.
- NG is the best single pruning method in terms of pruning, even better than all the other methods combined CDB-SB-RS.
- The Epilitis with options CDB-SB-RS-NG-10 considerably improves performance over all other methods combined CDB-SB-RS.

4.8.2 Heuristics and optimal no-good size bound

The second set of experiments has a factorial design. We defined three more heuristics ***H1, H2***, and ***H3*** and we ran each with all the pruning methods CDB-SB-RS-NG and a no-good size limit in the set {2, 4, 6, 10, 14, 18}.

The idea in the standard MRV (*Minimum Remaining Values*) heuristic is to choose the variable with the smallest domain so the current node has the smallest branching factor possible. This is a useful heuristic but not very informative since all domains have maximum size two, and so it is often the case that there are many ties. Additional search control guidance is definitely desired. The experiments of ACG [Armando, Castellini et al. 1999] show that significant gains in performance are made by breaking the ties in some informative non-arbitrary way. ACG chooses the variable that contains the value with the maximum pairwise inconsistencies with other remaining values. For example, if variable v has a value x that is inconsistent with 10 other values, then when variable v and value x are chosen, forward-checking will remove 10 values from the domains of the unassigned variables, maximally pruning the search space. The heuristic ***H0*** that we presented in the previous section would have chosen variable v to branch next.

For the value selection heuristic the opposite tactic is usually followed in the CSP literature: the value that least constrains the search is selected. In the previous example x would be the first value to be selected. The idea is that since v has to have a value assigned in any solution, we have more probabilities of finding a solution by assigning a value that constraints the search the least.

To summarize, the variable that constrains the search the most is selected first and the value that constrains the search the least is preferred to increase the probability of finding a solution as an extension of the current assignment.

In all of the heuristics we explored we followed the same principle. For the search heuristcs that we have tried the "variable that constrains the search the most" is selected according to some measure and the "value that constrains the search the least" is selected according to the *same* measure.

Let us define the *INC(p, V)* of a value *p* as the number of pairwise inconsistencies with between *p* and other values *q* in the domains of the variables in *V*. Also, we will use *NG-count(p)* to denote the no-goods the value *p* currently participates in. In the discussion below recall that *D(v)* and *d(v)* are the original and the current domain of variable *v*.

The heuristics we used are:

- ***H1***
  - Preorder the variables according to $max_{p \in D(V1)}$ *INC(p, V)* , where *V* is all DTP constraints (meta-CSP variables) and *V1* ∈ *V*. In other words, each variable *V1* is assigned the maximum of *INC(p, V)* over all of its values and then the variables are ordered according to this number. During search, select the variable according to the MRV (*Minimum Remaining Values*). Break the ties according to the preorder (thus breaking the ties is performed by a static heuristic).
  - Preorder in *ascending* order the values *p* ∈ *D(v)* in the domain of a variable *v* according to *INC(p, V)* , where *V* is all DTP constraints. During search use the preorder to pick the next value (thus, the value selection heuristic is static).
- ***H2***
  - Select the variable according to the MRV. Break the ties by selecting the variable $argmax_{v \in V}$ $max_{p \in d(V1)}$ *(INC(p, V) + NG-count(p))* , where *V* are the *unassigned variables*. In other words, we select the variable with the maximum $max_{p \in d(V1)}$ *(INC(p, V) + NG-count(p))*.
  - Select the value $argmin_{p \in d(V1)}$ *(INC(p, V) + NG-count(p))*.
- ***H3***
  - Select the variable according to the MRV. Break the ties by selecting the variable $argmax_{v \in V}$ $max_{p \in d(V1)}$ *INC(p, V)* , where *V* are the *unassigned variables*. If there are any more ties, break them by choosing $argmax_{v \in V}$ $max_{p \in d(V1)}$ *NG-count(p)*
  - Select the value $argmin_{p \in d(V1)}$ *INC(p, V)*. Break the ties by choosing $argmin_{p \in d(V1)}$ *NG-count(p)*.

H1 uses only the counts of *INC(p, V)* to statically predetermine an order of the variables and values that will break the ties both for variables and values. ***H0*** that we have used in the previous set of experiments, is very similar to ***H1*** but it break the ties using dynamic information, again by using the counts *INC(p, V)* but this time *V* is the remaining unassigned variables and not all the variables. For example, if a variable *v* has a value *x* that is inconsistent with 10 other values and this is the maximum number of inconsistencies for a value, ***H1*** will pick *v* . However, ***H0*** will pick *v* with certainty only if these values that are inconsistent with *x* are in unassigned variables, because if they belong to assigned variables, assigning *x* to *v* does not really constrain the space as much as the number 10 indicates.

H2 goes a step further and uses as heuristic information the recorded no-goods. By assigning $v \leftarrow x$ we not only constrain the search space by removing the values inconsistent with x, but we also make progress toward removing other values, namely those that participate in no-goods with x. For example, if we have the no-good $<\{ v \leftarrow x , q \leftarrow p , k \leftarrow l\}, J>$, then when x is assigned to v, we are one step closer to removing either p or l from the domains of q and k respectively. Thus ***H2*** uses the measure *INC(p, V) + NG-count(p)* to approximate how much a value p constrains the search space. ***H3*** is similar to ***H2*** but it only uses the no-good information *NG-count(p)* as secondary information, to break any ties that arise when using the measure *INC(p, V)*.

In the actual implementation ***H1*** takes time $O(mds \times |C|)$, where *mds* is maximum domain size to select the next variable and $|C|$ is the number of DTP constraints, since in the worst case it has to go through all the meta-CSP variables (DTP constraints) to select the one with the smallest domain (if the sizes of the domains are maintained every time we add or remove values from them, then the factor *mds* can be removed from the order of the time complexity). Breaking the ties in variable selection and performing value selection depends on a precomputed static ordering and so it takes constant time. All other heuristics ***H0, H2,*** and ***H3*** take $O(mds^2 |V|^2)$ time to select the next variable. This is because for every variable, and for every value in that variable, all other values in every other variable have to be checked for pairwise inconsistency. The numbers *NG-count(p)* take constant time to calculate since in our implementation they are cached when a no-good is recorded. Using incremental techniques we could drop the order of the brute force $O(mds^2 |V|^2)$ time algorithm but we have not implemented this in Epilitis.

In the graphs for this set of experiments we name the curves as "NG *x y*" to denote Epilitis with *all the pruning options on* and where x is the limit on the size of no-goods and y the heuristic used. If the heuristic is the default ***H0*** , y is omitted from the name. We just show the graphs for N=30 since we are interested in the asymptotic behavior of the heuristics and for smaller N the problems are too easy for Epilitis with all the options on.

The results are shown in Figure 4-25, Figure 4-26, and Figure 4-27. We are mostly concerned with the performance of each algorithm for the center of the critical region where *R=6* since this is where the difficult problems lie. As we can see for NG-2 the best heuristic is ***H1***. This is explained because the limit on the size of recorded no-goods is too tight for the no-goods to provide enough information to the heuristics ***H2, H3*** to balance the calculation overhead they impose. It is somewhat surprising however that ***H1*** does better than its dynamic counterpart ***H0***. It seems that our implementation of ***H0*** is too inefficient for the advantages of dynamically selecting the variable and value to outweigh the additional overhead. In fact, ***H1*** always does better than ***H0***[+] except for *R=5*.

For all other bounds on the no-good size that we have tried, ***H2*** or ***H3*** was always the winner. This supports the hypothesis that ***no-goods can successfully be used as heuristic information,*** apart from their use as pruning tools. For NG-6 the best heuristic is ***H2*** , for NG-14 it is ***H3*** , and for NG-10 and NG-18 the performances of ***H2*** and ***H3*** are similar. As a tiebreaker we decided to use the performance of the heuristics measured by the Average Time. The Average

[+] However, when we compared average times, ***H0*** was often better than ***H1*** indicating that it guides the algorithm to a "smoother" behavior with fewer outliers. So, we do not suggest completely dismissing ***H0*** without further investigation.

Time comparison is shown at Figure 4-28. Clearly, in NG-10, ***H2*** performs better than ***H3***, while this difference is marginal for NG-18.

Next, in order to find the best algorithm overall, we compared the winners of the previous comparisons together: NG-2-H1, NG-6-H2, NG-10-H2, NG-14-H3, and NG-18-H2. The results are shown in top of Figure 4-29 where the algorithms NG-2-H1 and NG-6-H2 are clearly losers. The remaining three algorithms are very close in performance and again as our tiebreaker we used their performance measured by the Average Time instead, which is shown at the bottom of the figure. The algorithm NG-10-H2, i.e. Epilitis with CDB, SB, RS, NG on, maximum no-good size 10, and heuristic ***H2***, is marginally better and will be selected as our best algorithm.

To summarize the results of this section:

- Epilitis with CDB, SB, RS, NG on, maximum no-good size 10, and heuristic ***H2*** is the best algorithm in the set of experiments we ran.

4.8.3 The number of forward-checks is the wrong measure of performance

It is customary in the literature to compare DTP solvers and report their performance using the number of forward-checks, more precisely the total number of values that the algorithm forward-checked during search, also called consistency checks *CCs*[18] [Stergiou and Koubarakis 1998; Armando, Castellini et al. 1999; Oddi and Cesta 2000; Stergiou and Koubarakis 2000]. This is because it is conjectured *CCs* is a good predictor of the actual time performance of the solver and moreover, it has the desirable characteristic that it is machine independent.

The conjecture has its roots in the CSP literature and it is justified for two reasons:

1. A consistency check is where a typical CSP solver spends most of its time.
2. For binary constraints in most CSP solvers, a consistency check of a constraint takes constant time and it is just a table lookup.

In DTP solving the above two conditions do not hold: a DTP solver spends a significant amount of time in **maintain-consistency** and in the variable and value heuristic and not only forward-checking. Also, a consistency check does not take the same time in all DTP solvers. As we display in Table 4-2, forward-checking a value (without no-good learning) takes time

1. $O(|V|)$, if we maintain directional path consistency (as in SK)
2. $O(1)$, if we maintain full path consistency (as in OC and Epilitis)
3. $O(|V|^3)$, if we do not propagate the constraints and check consistency from scratch (as in ACG)

In addition, when no-goods are recorded and used during forward checking, then additional time is required to check the value. It becomes obvious then, that *CCs* is the wrong machine and implementation independent measure of performance. ***We suggest an alternative comparative machine independent measure of performance between two algorithms that use the same variable and value heuristics***. Our suggestion is the statistic S = T(**maintain-consistency**) +

[18] In our implementation a consistency-check is essentially checking the FC-condition. We also felt that we should count as a consistency-check determining whether the Subsumption-Condition holds, because both are similar operations and take the same time. Not counting the Subsumption-Condition checks in *CCs* would favor all algorithms with RS on since those are the ones that perform this operation.

T(**forward-check**), where T(**maintain-consistency**) is the total time spent in function **maintain-consistency** and T(**forward-check**) the total time spent in function **forward-check**.

In Epilitis, it is easy to estimate *S* when no-good recording is off. The algorithm we use to implement **maintain-consistency** takes time $\Theta(|V|^2)$ per call. Thus the total time in **maintain-consistency** is $|V|^2$ *Cprops* (see the beginning of Section 4.8 for the definition of *Cprops* and *CCs*).. The time to forward-check a value takes O(1) time since we maintain full path consistency. So, the total time in **forward-check** is equal to the number of consistency-checks *CCs*. Thus, we estimate *S* as $S = |V|^2$ *Cprops* + *CCs* for Epilitis without no-good recording (on a DTP with $|V|$ variables). The statistics *Cprops* and *CCs* are included in the output of Epilitis.

We would also like to add that the number of *Cprops* (i.e. calls to **maintain-consistency**) is not the same as *Nodes*, i.e. the number of nodes generated during search, since in a CSP-based DTP solver **maintain-consistency** might be called multiple times in a node to propagate the semantic-branching constraints.

It is also important to notice that when only *CCs* are reported, a DTP solver can "cheat" by decreasing the number of *CCs* while increasing the time in other computations. One such technique is FC-off that we described at Section 4.3.4. Recall that in FC-off it is possible to reduce the number of *CCs* by generating additional nodes and calls to **maintain-consistency**. In [Stergiou and Koubarakis 2000] versions of their algorithms with FC-off were tested and compared to other ones using only *CCs*.

We now provide evidence to support our hypothesis that *S* is a better predictor of performance than *CCs*. Table 4-8 shows all the algorithms with no-good recording off that we ran on problems with size N=30. The first column orders the algorithms in increasing order according to their performance in the critical region where R=6 measured by the average time. The second column orders the algorithms by using the average number of *CCs* and the third by our estimation of the average *S* statistic we described above. It is easy to see that the average *CCs* favors the algorithms that use FC-off (denoted with an FC in their name) and ranks them the best while in fact they are the worst in terms of average time performance. However *the S statistic correctly orders all algorithms* with one mistake: the order of SB-RS and CDB-SB is reversed.

We repeated the same procedure as above for N=20 for which we have the results from more variations of Epilitis. The results are displayed in Table 4-9. We can see that *S* is again a very good predictor of the time: the two orderings are the same with three minor differences: CDB-RS-FC is placed two positions higher, and the pairs (CDB-SB-RS-FC, SB-FC) and (SB-RS, CDB-SB) appear in the reverse order. Especially the latter two mistakes are localized. In contrast *CCs* does not order the algorithms correctly, e.g. CDB-SB-RS-FC is ranked the best, five positions higher than it really is.

Even though we have not displayed a complete analysis of algorithms with FC-off, it is easy to see from the tables that we have given that invariably FC-off degrades the performance.

In contrast with the above analysis, Stergiou and Koubarakis note "We have also measured the CPU times used by the algorithms we studied. As expected, the CPU times are proportional to the number of consistency checks", end of section 6.2, [Stergiou and Koubarakis 2000]. We hypothesize that this is because in their implementation each consistency-check was not a simple array lookup. Instead, it involved one constraint propagation and one constraint retraction. Thus, the total time spent in **forward-check** greatly dominates the time spent in **maintain-consistency**.

We also attempted to develop a predictor for the performance of no-good recording algorithms too by incorporating the number of no-good checks *NCs* (see the beginning of Section 4.8) in the model and approximate *S* by $S = |V|^2 \, Cprops + CCs + NCs$. Unfortunately, this method did not prove a good predictor of the actual time performance.

Summarizing the results of this section:

- We provided evidence that the currently accepted machine independent measure of performance for DTP solvers, i.e. the count of consistency-checks *CCs*, is not accurate.
- We developed a different measure *S* = T(**maintain-consistency**) + T(**forward-check**). We showed how to approximate it for Epilitis (without no-good recording) as $S = |V|^2 \, Cprops + CCs$, and showed that it is an accurate predictor of performance of Epilitis.
- We showed that the technique FC-off used in some DTP solvers can degrade the performance of the solver.

4.8.4 Comparing Epilitis to the previous state-of-the-art DTP solver

In this set of experiments we compare Epilitis CDB-SB-RS-NG-10-H2 with the previous state-of-the-art DTP solver by ACG (also called TSAT). We depart from the common practice in the literature to use the median number of *CCs* for such a comparison. We explained the reasons why we believe this is the wrong comparison measure in Section 4.8.3. Unfortunately, we could not use our improved comparison statistic because the output of ACG's solver does not include the number of *Cprops*. In addition, our statistic is not accurate for Epilitis versions with no-good recording. Thus, we believe that the only meaningful comparison is directly comparing times. One drawback of such a comparison is that we cannot as easily draw conclusions about the pruning efficiency of Epilitis additional pruning methods, such as RS, CDB, and NG, and the efficacy of heuristic ***H2***, since ACG uses a very inefficient method for consistency checks. Thus, the better performance of Epilitis can be only attributed to the better consistency checking techniques it employs. We cannot totally dismiss this hypothesis until a better machine-independent and implementation-independent predictor of performance is developed for algorithms with no-good recording and ACG release a version of their solver that reports additional descriptive statistics (e.g. nodes generated).

Figure 4-30 shows the graphs for N=25 and N=30. Epilitis is faster by about two orders of magnitude for the larger (N=30) problems. Also recall that ACG have a time-out of 1000 seconds imposed on their solver and so the real median time taken by their program might be significantly more than it is indicated here. For example, for N=30 and R=6 the median time is exactly 1000 seconds implying that their solver timed-out on more than half the problems.

We also ran the best version of Epilitis on larger problems where N=35 and notice that the algorithm scales well. Figure 4-31 shows the performance of Epilitis on problems of different sizes. For the larger problems N=35 Epilitis has a median time performance of about 100 seconds. The overall performance of TSAT is also shown in the same figure. As we can see TSAT requires about one order of magnitude more time every time N is increased. In contrast Epilitis' performance does not degrade as fast.

Summarizing the results of this section:

- Epilitis is almost two orders of magnitude faster than the previous state-of-the-art DTP solver TSAT.
- Epilitis' performance scales comparatively well as the size of the problems increase.

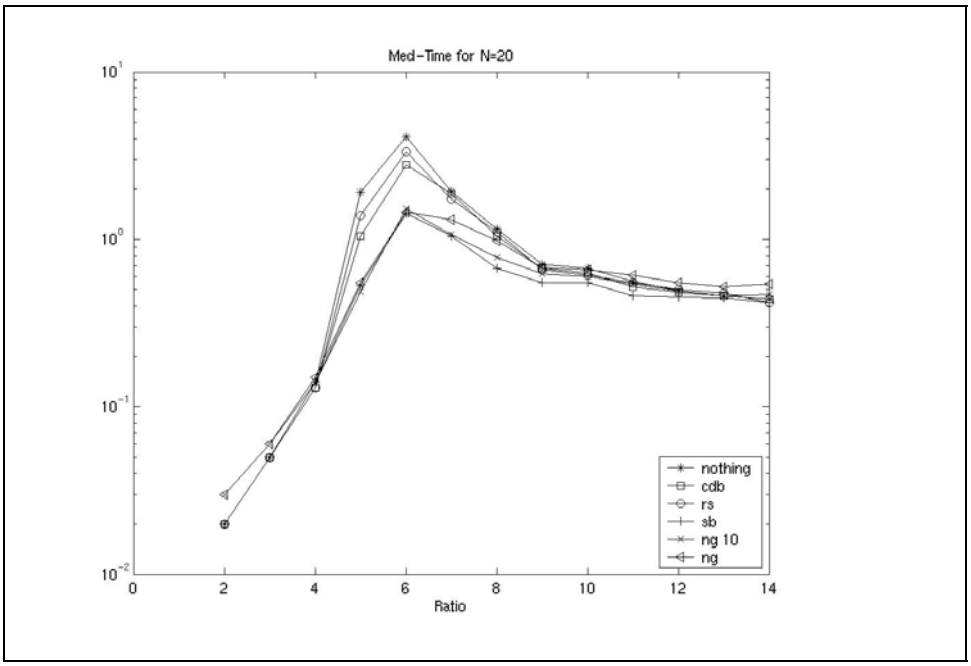


Figure 4-21: Comparing single pruning methods , N-20.

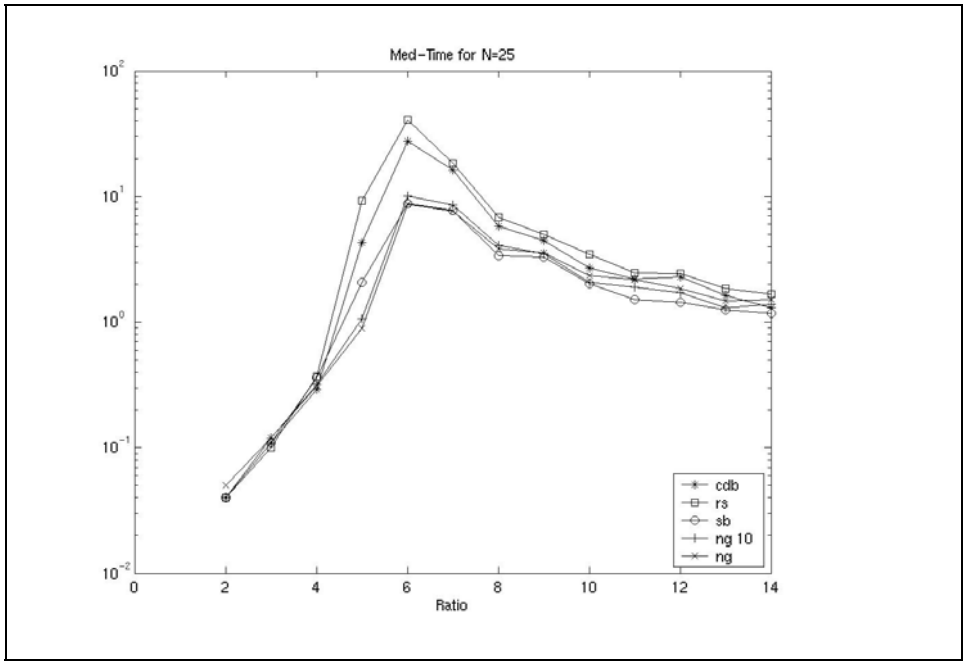


Figure 4-22: Comparing single pruning methods , N-25.

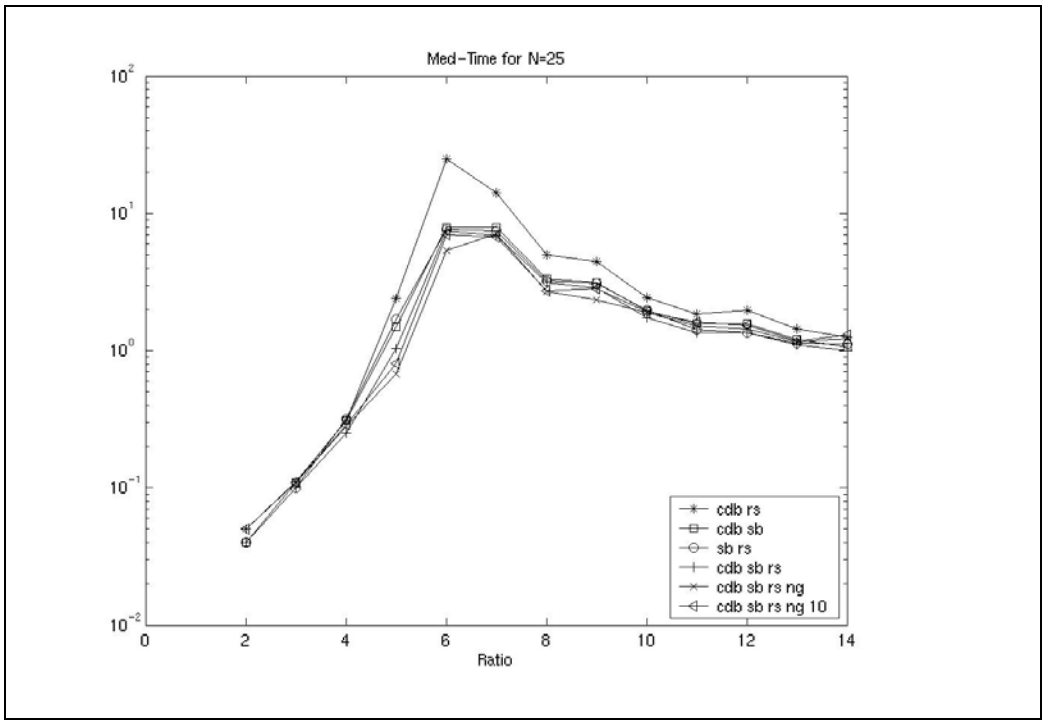


Figure 4-23: Comparing combinations of pruning methods, N=25

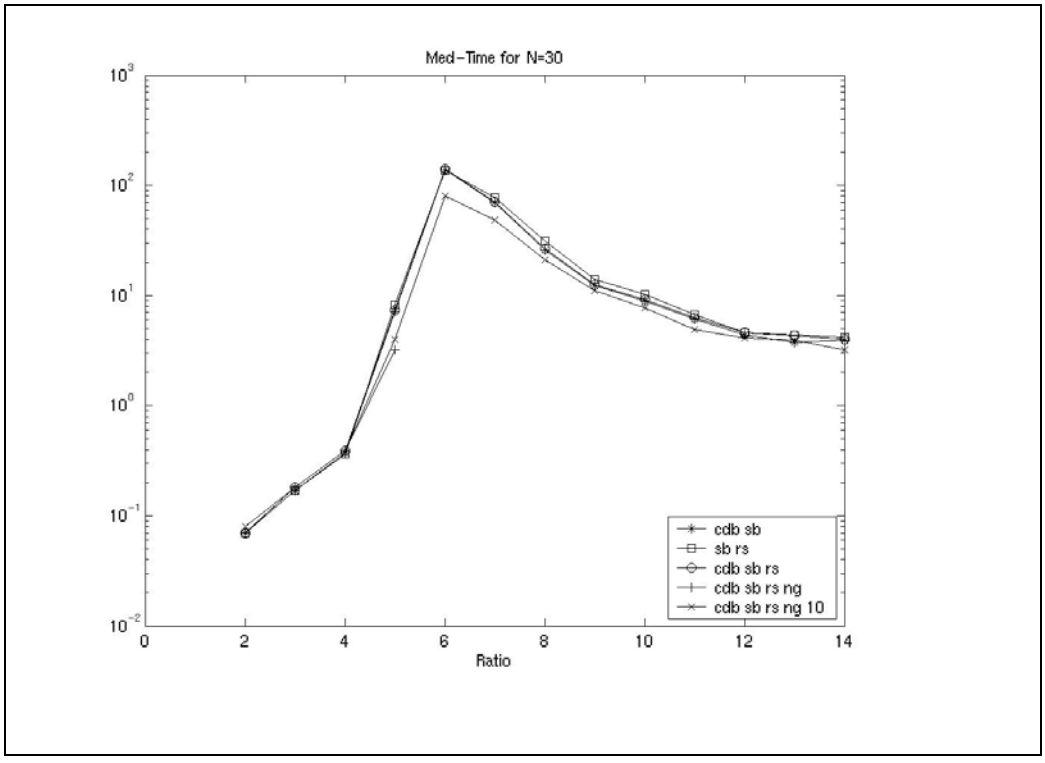


Figure 4-24: Comparing combinations of pruning methods, N=30

Ratio	nothing	rs	cdb	cdb rs	ng_10	ng	sb	sb rs	cdb sb	cdb sb rs	cdb sb rs ng	cdb rs sb ng 10
2	0.02	0.02	0.02	0.02	0.03	0.03	0.02	0.02	0.02	0.02	0.03	0.021
3	0.05	0.05	0.05	0.05	0.06	0.06	0.05	0.05	0.05	0.05	0.06	0.06
4	0.14	0.13	0.13	0.11	0.14	0.15	0.14	0.12	0.14	0.11	0.13	0.13
5	1.91	1.39	1.05	0.811	0.49	0.551	0.531	0.51	0.501	0.451	0.421	0.431
6	4.1	3.33	2.8	2.39	1.53	1.46	1.43	1.34	1.25	1.24	1.04	0.941
7	1.93	1.74	1.87	1.5	1.07	1.31	1.05	1.01	0.981	0.971	1.02	0.851
8	1.15	1.11	1.05	0.892	0.781	0.982	0.671	0.651	0.661	0.621	0.751	0.701
9	0.711	0.661	0.671	0.611	0.621	0.681	0.551	0.54	0.521	0.53	0.62	0.571
10	0.671	0.611	0.631	0.571	0.6	0.66	0.55	0.551	0.541	0.511	0.571	0.531
11	0.56	0.551	0.521	0.521	0.541	0.61	0.461	0.441	0.451	0.441	0.531	0.511
12	0.491	0.501	0.481	0.461	0.491	0.551	0.451	0.431	0.43	0.431	0.521	0.48
13	0.461	0.48	0.461	0.461	0.461	0.521	0.45	0.44	0.42	0.411	0.51	0.48
14	0.441	0.42	0.441	0.43	0.471	0.541	0.42	0.41	0.421	0.411	0.551	0.491

Table 4-3: The ordering of the pruning methods according to Median Time when R=6, and N=20.

Ratio	rs	cdb	cdb rs	ng_10	sb	ng	cdb sb	sb rs	cdb sb rs	cdb rs sb ng_10	cdb sb rs ng
2	0.04	0.04	0.05	0.04	0.04	0.05	0.04	0.04	0.04	0.05	0.05
3	0.1	0.11	0.111	0.12	0.11	0.12	0.11	0.1	0.1	0.11	0.11
4	0.37	0.291	0.31	0.32	0.361	0.311	0.311	0.32	0.251	0.29	0.281
5	9.22	4.3	2.41	1.06	2.08	0.892	1.5	1.69	1.04	0.811	0.681
6	40.8	27.5	24.8	10.1	8.77	8.72	7.98	7.7	7.43	7.05	5.38
7	18.5	16.3	14.1	8.56	7.72	7.66	7.94	7.47	6.94	6.78	7.09
8	6.8	5.81	5.02	4.1	3.37	3.83	3.36	3.2	3.14	2.74	2.71
9	4.99	4.46	4.45	3.51	3.3	3.56	3.14	3.1	2.89	2.83	2.36
10	3.46	2.69	2.45	2.08	2.01	2.36	1.94	1.97	1.74	1.92	1.9
11	2.48	2.21	1.86	1.9	1.52	2.17	1.57	1.42	1.36	1.61	1.52
12	2.44	2.3	1.98	1.71	1.44	1.86	1.58	1.36	1.36	1.53	1.44
13	1.84	1.63	1.44	1.31	1.26	1.48	1.2	1.13	1.11	1.18	1.16
14	1.68	1.3	1.25	1.38	1.18	1.51	1.07	1.12	1	1.3	1.22

Table 4-4: The ordering of the pruning methods according to Median Time when R=6, and N=25.

Ratio	sb	cdb sb rs	cdb sb	sb rs	cdb sb rs ng_10
2	0.07	0.07	0.07	0.07	0.08
3	0.17	0.18	0.17	0.17	0.18
4	0.4	0.39	0.371	0.36	0.39
5	10.3	7.42	7.12	8.25	4
6	149	142	140	138	79.8
7	74.6	70.5	69.7	78.2	48.8
8	29.4	26.6	25.9	31.5	21.1
9	13.9	12.6	12.4	14.1	11.1
10	9.73	9.15	8.92	10.3	7.72
11	6.73	6.28	6.09	6.69	4.93
12	4.47	4.64	4.42	4.61	4.12
13	4.32	4.31	3.77	4.38	3.97
14	3.96	4.02	3.91	4.17	3.2

Table 4-5: The ordering of the pruning methods according to Median Time when R=6, and N=30

Ratio	rs	cdb	cdb rs	sb	sb rs	cdb sb	cdb sb rs	ng_10	ng	cdb sb rs ng_10	cdb sb rs ng
2	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
3	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
4	100.00%	100.00%	98.91%	97.83%	97.83%	97.83%	97.83%	100.00%	100.00%	97.83%	97.83%
5	76.00%	42.00%	42.00%	30.40%	30.40%	27.73%	27.73%	20.33%	17.53%	15.67%	15.13%
6	88.84%	67.36%	63.64%	35.29%	35.29%	32.77%	32.52%	31.12%	27.31%	19.75%	19.75%
7	89.71%	79.22%	73.69%	50.49%	50.49%	45.24%	44.66%	39.13%	39.13%	31.17%	32.82%
8	100.00%	84.01%	77.07%	54.72%	54.72%	51.25%	50.67%	52.99%	53.95%	44.51%	44.12%
9	91.20%	84.00%	84.00%	78.00%	78.00%	72.00%	72.00%	70.80%	70.00%	59.60%	59.60%
10	95.24%	81.43%	78.10%	77.62%	77.62%	69.05%	69.05%	68.10%	68.10%	56.19%	56.19%
11	98.57%	81.43%	81.43%	77.14%	77.14%	67.86%	67.86%	71.43%	71.43%	66.43%	66.43%
12	100.00%	96.12%	96.12%	95.15%	95.15%	86.41%	86.41%	84.47%	84.47%	78.64%	78.64%
13	100.00%	88.64%	88.64%	92.05%	92.05%	81.82%	81.82%	84.09%	84.09%	79.55%	79.55%
14	100.00%	97.56%	97.56%	95.12%	95.12%	87.81%	87.81%	82.93%	82.93%	82.93%	82.93%

Table 4-6: The statistic Median Nodes divided by Median Nodes of "Nothing" for N=20. The pruning methods are sorted according to this statistic for R=6.

Ratio	sb	sb rs	cdb sb	cdb sb rs	cdb sb rs ng 10
2	100.00%	100.00%	100.00%	100.00%	100.00%
3	100.00%	100.00%	100.00%	100.00%	100.00%
4	100.00%	89.31%	100.70%	99.30%	100.00%
5	100.00%	92.96%	76.77%	97.04%	46.44%
6	100.00%	100.00%	94.93%	100.00%	41.07%
7	100.00%	100.00%	97.13%	95.78%	50.66%
8	100.00%	100.00%	89.59%	100.00%	62.39%
9	100.00%	100.00%	86.26%	100.00%	77.43%
10	100.00%	100.00%	93.37%	100.00%	69.68%
11	100.00%	100.00%	85.98%	100.00%	70.14%
12	100.00%	100.00%	92.95%	100.00%	68.39%
13	100.00%	100.00%	95.23%	100.00%	76.20%
14	100.00%	100.00%	96.97%	100.00%	68.75%

Table 4-7: The percentage of search pruning, for N=30.

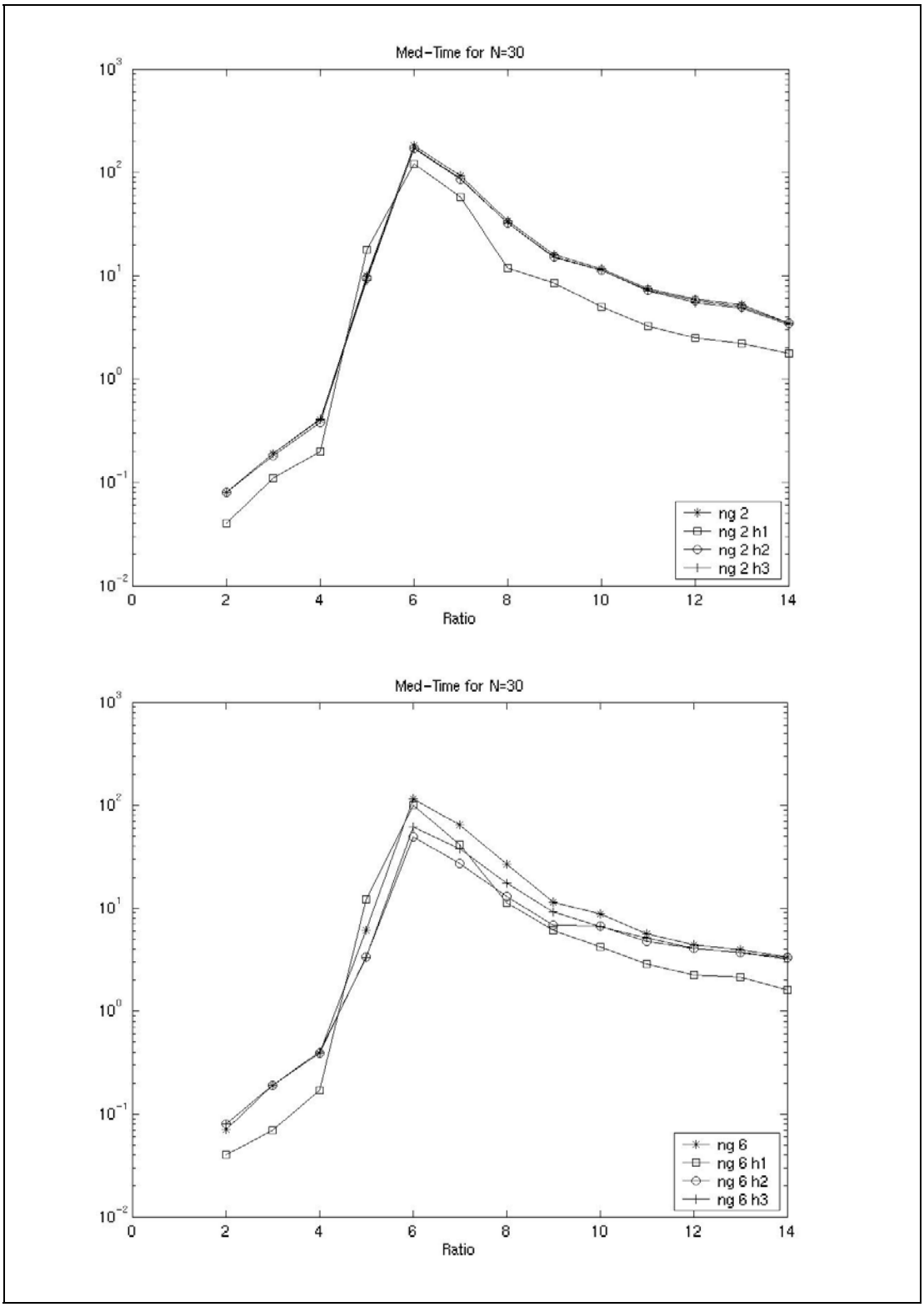


Figure 4-25: Comparison of different heuristics for N=30 and algorithms NG=2 and NG=6.

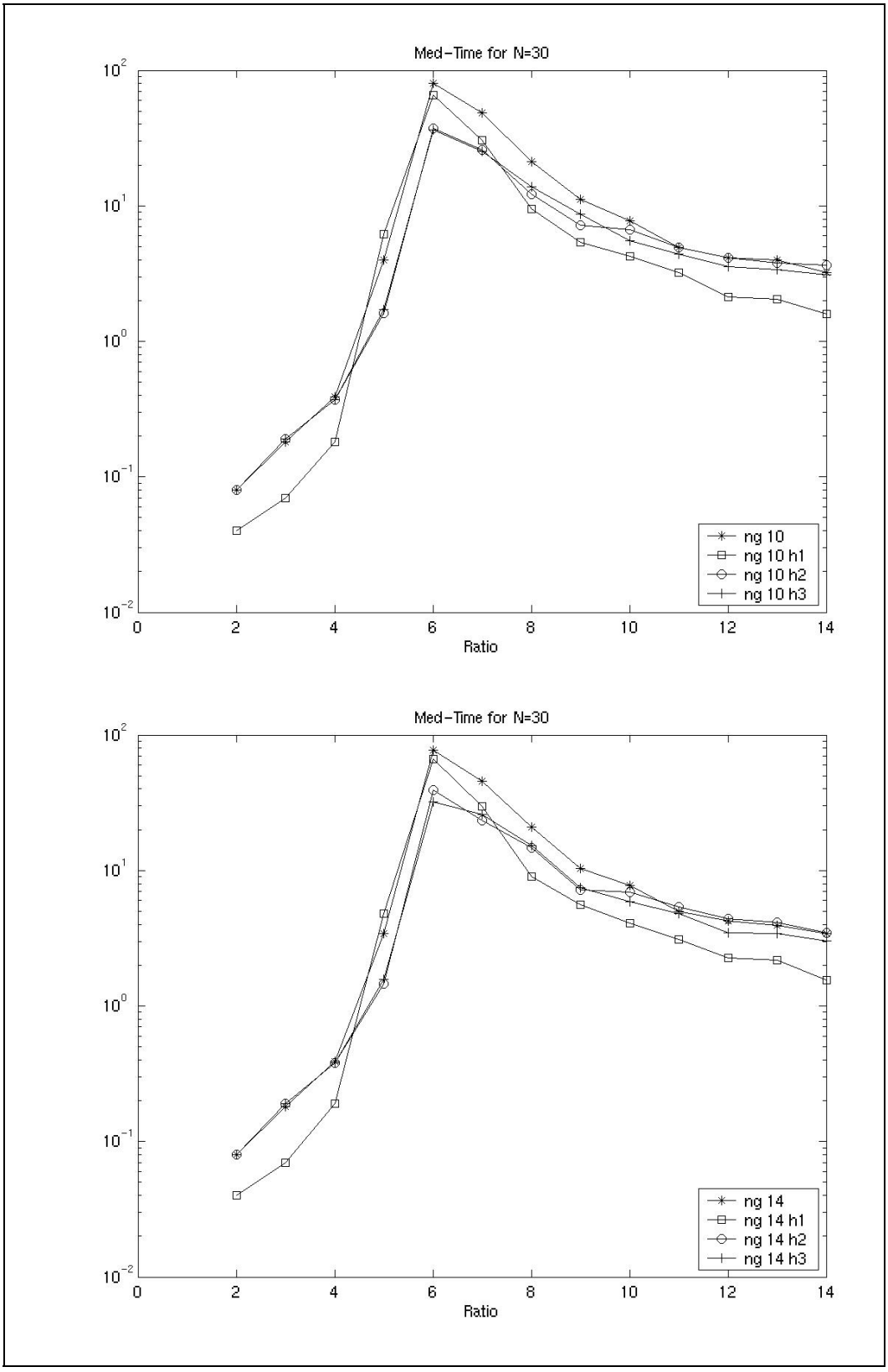


Figure 4-26: Comparison of different heuristics for N=30 and algorithms NG=10 and NG=14.

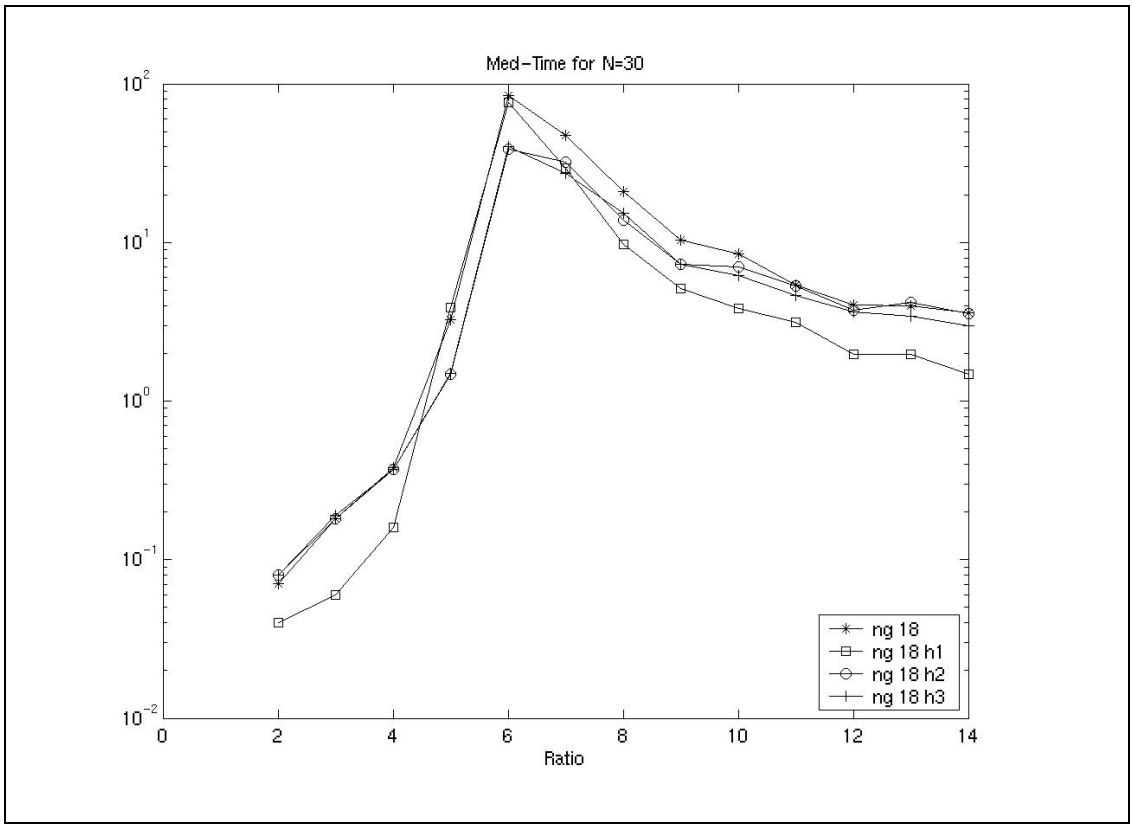


Figure 4-27: Comparison of different heuristics for N=30 and algorithms NG=18.

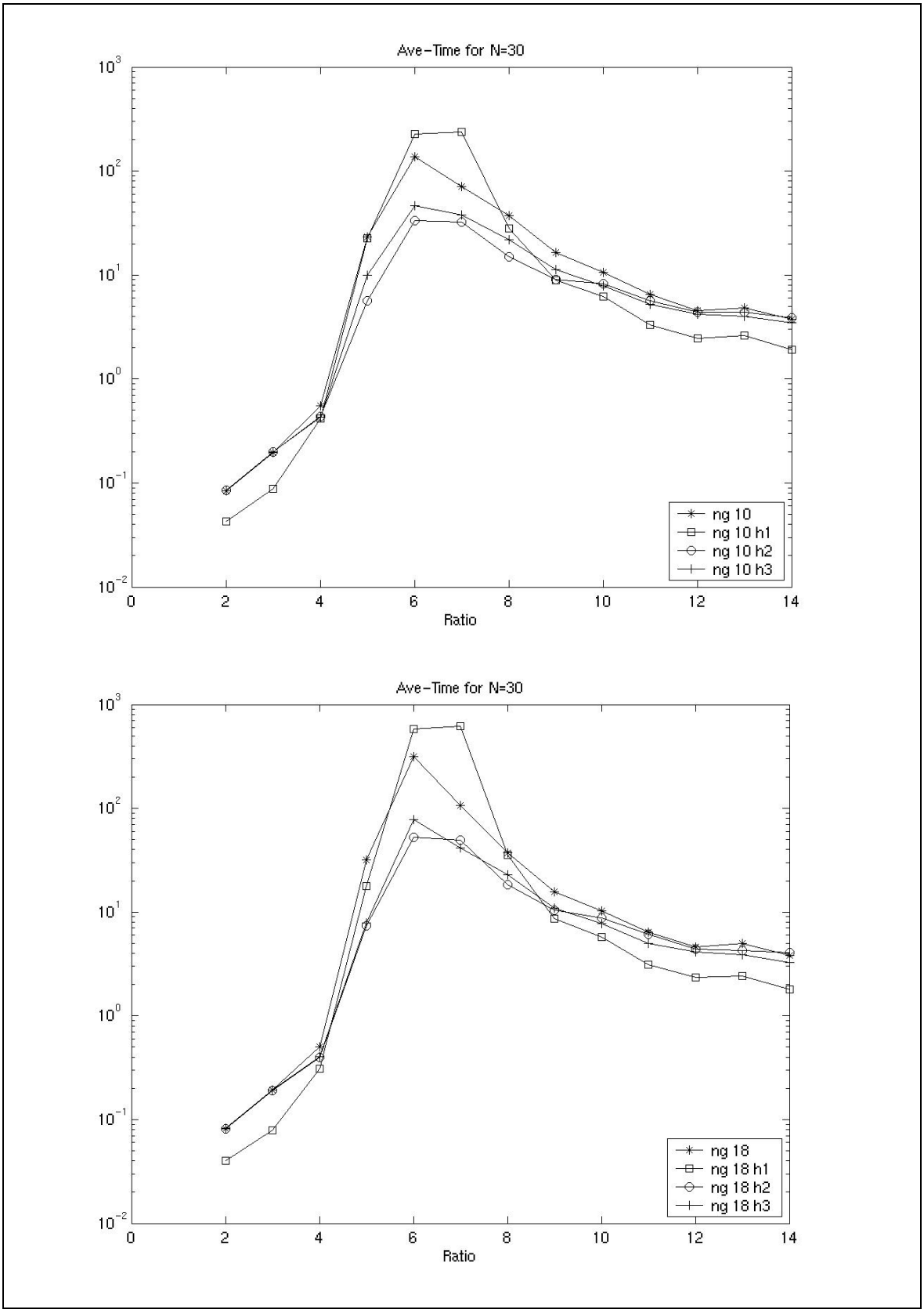


Figure 4-28: Comparison of Average Time of different heuristics for N=30, for algorithms NG=10 and NG=18.

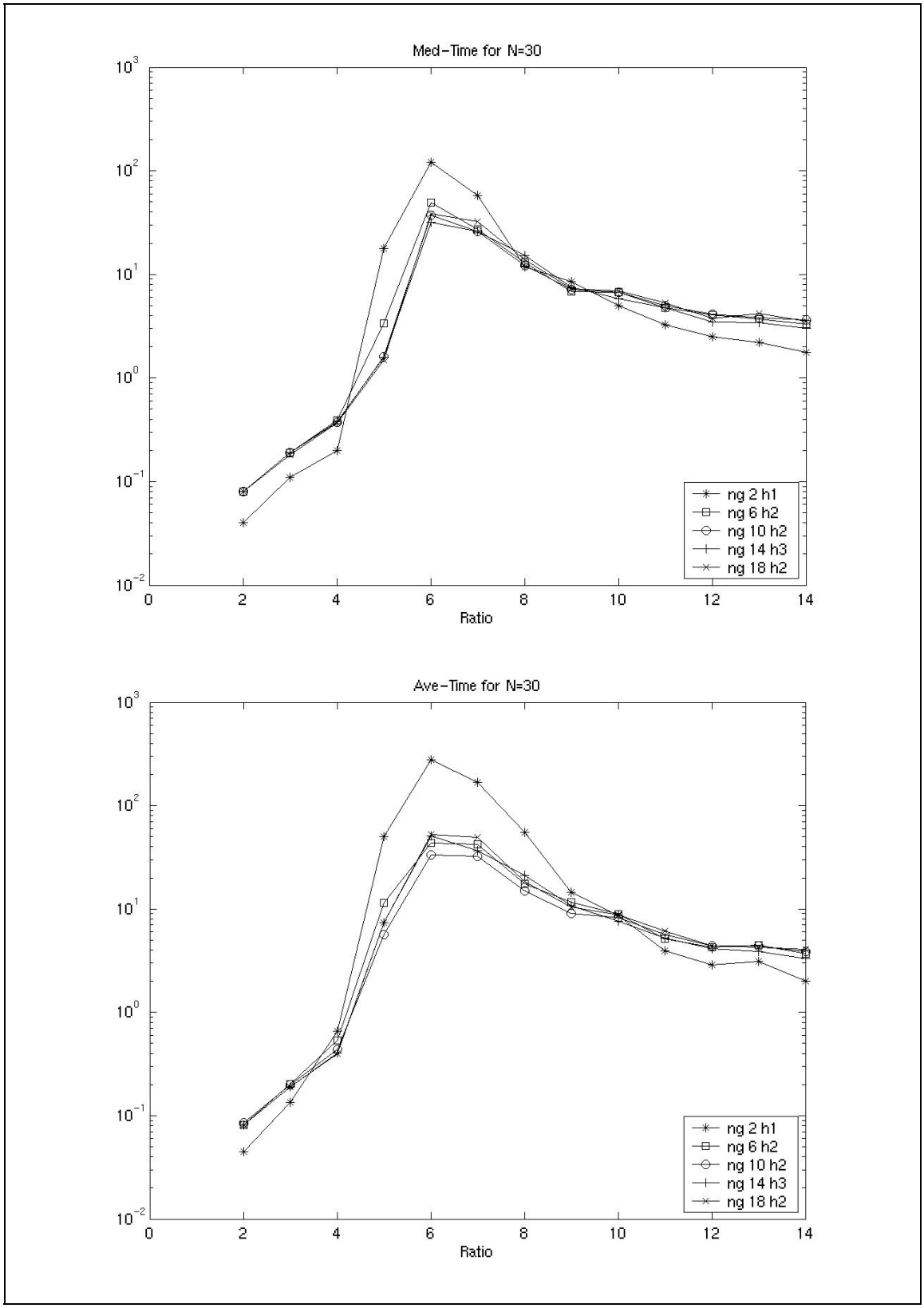


Figure 4-29: Comparison for the best overall algorithm for N=30.

N=30		
Ave-Time	**Ave-CCs**	**Ave-S**
SB-FC	SB	SB-FC
CDB-SB-RS-FC	CDB-SB	CDB-SB-RS-FC
SB	SB-RS	SB
SB-RS	CDB-SB-RS	CDB-SB
CDB-SB	SB-FC	SB-RS
CDB-SB-RS	CDB-SB-RS-FC	CDB-SB-RS

Table 4-8: The ordering of performance of algorithms according to N=30, R=6 for the average of various statistics.

N=20		
Ave-Time	**Ave-CCs**	**Ave-S**
Nothing	Nothing	Nothing
CDB-FC	RS	CDB-FC
CDB-RS-FC	CDB	RS
RS	CDB-RS	CDB
CDB	CDB-FC	CDB-RS-FC
CDB-RS	CDB-RS-FC	CDB-RS
CDB-SB-RS-FC	SB	SB-FC
SB-FC	CDB-SB	CDB-SB-RS-FC
SB	SB-RS	SB
SB-RS	CDB-SB-RS	CDB-SB
CDB-SB	SB-FC	SB-RS
CDB-SB-RS	CDB-SB-RS-FC	CDB-SB-RS

Table 4-9: The ordering of performance of algorithms according to N=20, R=6 for the average of various statistics.

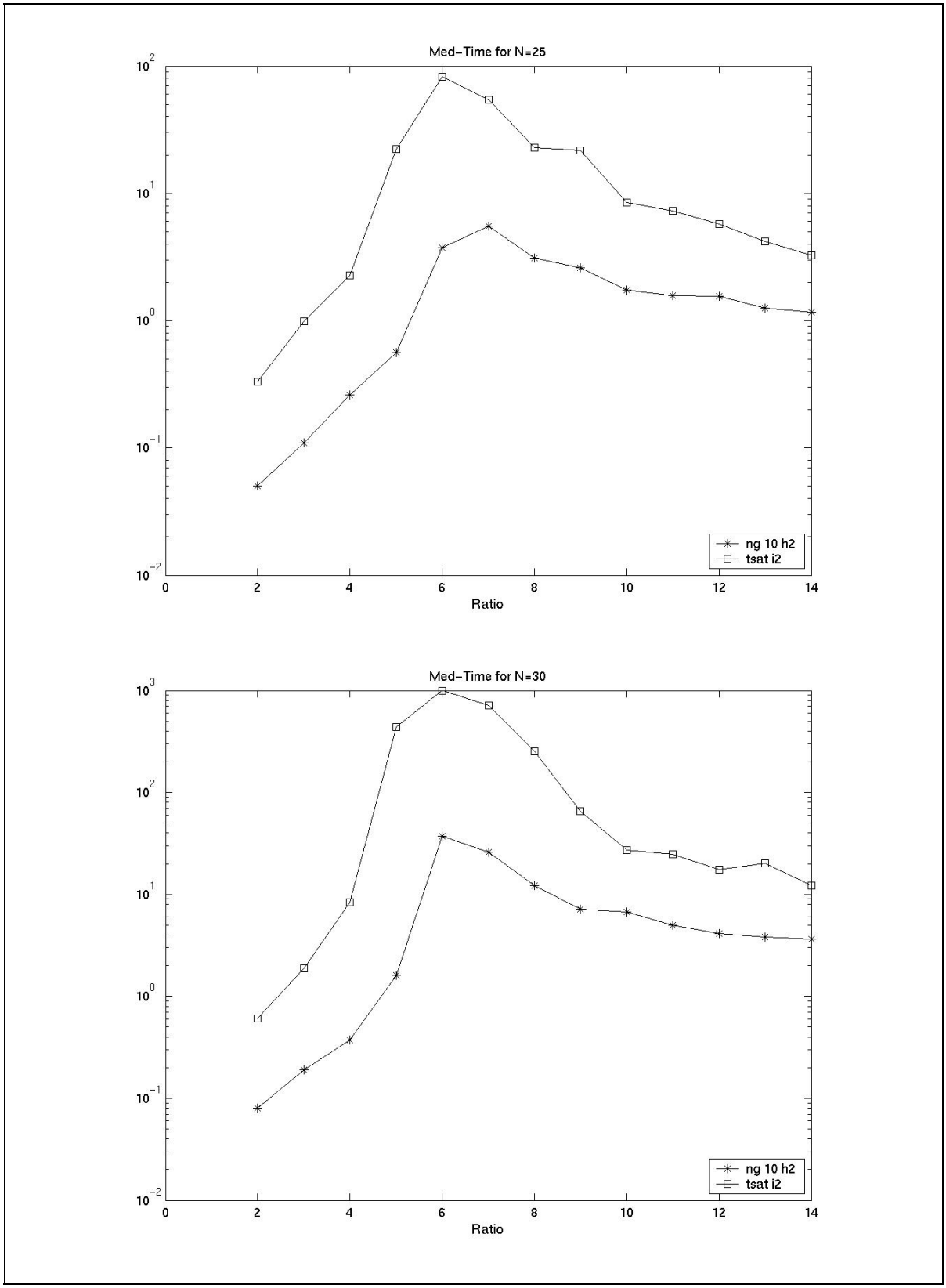


Figure 4-30: Comparison of Epilitis and ACG's solver for N=25 and N=30.

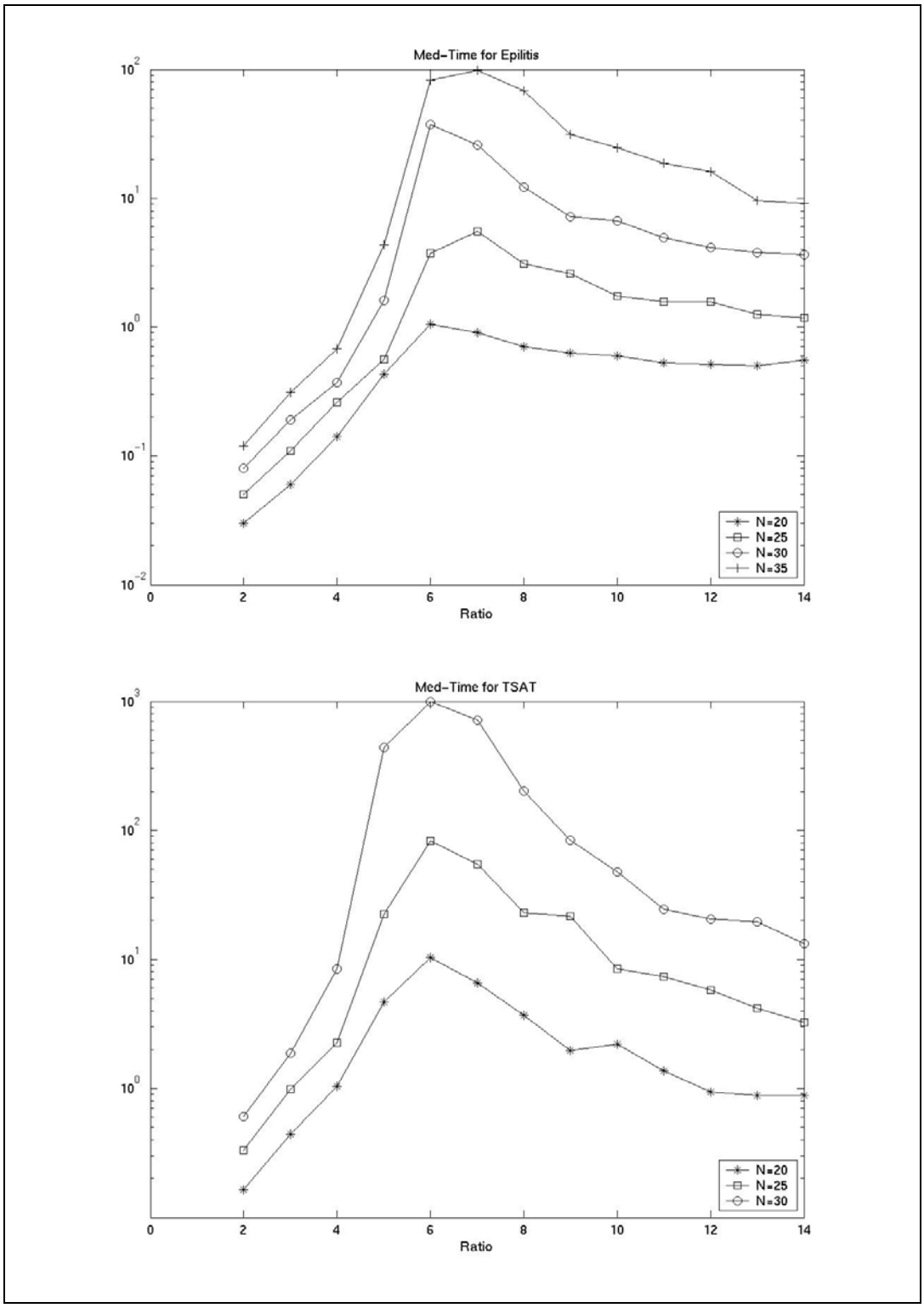


Figure 4-31: The median time performance of Epilitis and TSAT.

4.9 Related Work

4.9.1 Similar Temporal Reasoning formalisms

We have already talked about DTP solvers in Section 4.7 . We now consider a broader context. DTP is a very expressive class of problems that subsumes a number of other formalisms. Obviously, the DTP subsumes the TCSP since the latter one allows constraints of the form: $c_{i1} \vee \ldots \vee c_{in}$ where each c_{ij} is of the form $x - y \leq b$ with the restriction that x and y are the same in all c_{ij} , while in DTP this restriction is removed. Thus, the TCSP constraints are binary, involving only two variables, while the DTP constraints are non-binary. This makes a significant difference in solving methods, since it is easier to calculate path-consistency in networks of binary constraints, than in networks where the constraints are non-binary [Bessiere and Regin 1997; Bessiere 1999; Bessiere, Mesequer et al. 1999]. In addition, the constraints in the meta-CSP are not explicit. Consequently, path-consistency and variations of it [Schwalb and Dechter 1998] are heavily used either for preprocessing or for forward checking during backtrack search in TCSPs, but they have not been employed not in DTP solving yet. Another factor for not using path-consistency in DTP solving is that the constraint satisfaction community has typically been relying on transforming the non-binary constraints to binary constraints [Bacchus and Beek 1998; Stergiou and Walsh 1999] and has relatively been neglecting constraint satisfaction techniques that apply directly to non-binary constraints [Bessiere 1999].

We consider the most serious disadvantage of TCSPs to be their inability to express the fact that two intervals should not overlap. This kind of constraint is essential in scheduling and planning applications where it is typically the case that some actions should not overlap each other, e.g. two actions that utilize the same resource. More specifically, suppose that A and B are two actions that use the same printer and thus A can precede B or vice-versa but they should not overlap. If we denote with A_S and B_S the start time-points of these actions and correspondingly A_E and B_E then the fact that A and B cannot overlap can be written as the constraint:

$A_E \leq B_S$ (A finishes, then B starts) or $B_E \leq A_S$ (B finishes, then A starts), or equivalently

$$A_E - B_S \leq 0 \vee B_E - A_S \leq 0 \text{ , in DTP format}$$

The constraint involves four time-points A_S , B_S , A_E , and B_E and so it cannot be represented by a TCSP, but it is perfectly fine in a DTP. There are other, *binary*, representations however, that allow such constraints to be represented. These include the *Point-Interval-Algebra* PIA described in [Meiri 1991]. In PIA the variables can be either time-points or intervals; and all relations are binary: interval-interval, point-point, inteval-point. The constraints between time-points can be metric TCSP constraints, while the rest of constraints are qualitative. Having two more interval variables A_I and B_I representing the intervals associated with the actions can then encode the above situation by imposing the disjunctive constraint:

$$A_I \{before, after\} B_I$$

Although PIA can thus model prohibited overlaps, it cannot readily handle requirements of temporal separation between actions. For example, suppose that A and B are two medical

treatment procedures applied to the same patient with the constraint that if A is applied first, B can only be applied 3 days later, while if B is performed first then A can be performed 2 days later. The constraint cannot represented in PIA but written as the DTP constraint

$$A_E + 3 \leq B_S \text{ or } B_E + 2 \leq A_S, \text{ equivalently}$$
$$A_E - B_S \leq -3 \vee B_E - A_S \leq -2$$

can be encoded as a DTP. Nevertheless, extensions of the PIA have appeared that allow constraints of this sort to be represented while remaining within the realm of binary constraints [Cheng and Smith 1995; Cheng and Smith 1995].

The above arguments may suggest that we can avoid non-binary constraints if we employ both intervals and time-points as our representational elements. There are common situations however, when there is no way that we can represent the temporal information without employing non-binary constraints. Consider the following example from [Dechter, Meiri et al. 1991]:

"*John goes to work either by car [30' – 40'], or by bus (at least 60'). Fred goes to work either by car [20' – 30'], or in a carpool [40'-50']. Today John left home (t1) between 7:10 and 7:20, and Fred arrived (t4) at work between 8:00 and 8:10. We also know that John arrived (t2) at work about 10'-20' after Fred left home (t3).*"

The story can be represented by a TCSP. Let us concentrate on the events of "John leaving home" (*t1*) and the events of "John arriving at work" (*t2*). Since John may go to work either by car or by bus, the TCSP-form constraint that holds between them is:

$$30 \leq t2 - t1 \leq 40 \vee 60 \leq t2 - t1 \leq \infty$$

Notice that the intervals of John taking the bus and John taking the car end the same time and coincide with the time John arrives at work so they can all be represented with the same variable *t2*. This is exactly what allows us to represent this disjunction as a binary constraint. If the story changes slightly, so that John after taking the bus, gets off in a nearby grocery store and spends at least five minutes buying milk (he leaves at time-point *t5*), then the time of which John arrives at work if he takes the bus (and buys milk) does not coincide with the time at which he arrives at work if he drives. We will use *t2a* for the time-point of John getting at work by car, and *t2b* for John getting off the bus, and *t2c* for the time John actually arrives at work. Then the modified story is represented as:

$$(30 \leq t2a - t1 \leq 40 \wedge 0 \leq t2c - t2a \leq 0) \vee$$
$$(60 \leq t2b - t1 \leq \infty \wedge 5 \leq t5 - t2b \leq \infty \; 0 \leq t2c - t5 \leq 0),$$

or in English, he either takes the car for 30 to 40 minutes and he gets at work when is done using it, or he takes the bus for at least an hour, spends at least five minutes in the store, and then he gets at work. The modified story is not represented by any formalism that only involves binary-constraints and relations between time-points, intervals, or both. It can be represented by DTP constraints by converting the above constraint in CNF form. As we will see, stories of this type motivate Conditional Simple Temporal Problems (CSTP) and in Chapter 6 we will see that consistency in CSTPs can be cast as a DTP problem.

Other types of constraints that are inherently non-binary are if-then constraints of the form "if *constraint1* then *constriant2* ", e.g. "if treatment A does not last enough, then perform treatment B for at least e days." The constraint can be written as:

$$\neg (d \leq A_E - A_S) \Rightarrow (e \leq B_E - B_S), \text{ or equivalently}$$
$$(d \leq A_E - A_S) \vee (e \leq B_E - B_S)$$

This desire to be able to express this type of constraints was also expressed by Meiri in [Meiri 1996]: "Future research should enrich the representation language to facilitate modeling of more involved reasoning tasks. In particular, non-binary constraints (for example `If John leaves home before 7:15am he arrives at work before Fred')" referring to the example quoted above.

There are two other formalisms that are as expressive as DTPs, namely the Generalized Temporal Network (GNC) described in [Staab 1998] and the Temporal Constraint Networks (TCN) of Barber [Barber 2000]. The former is essentially DTPs with disjuncts that allow conjunctions of STP-like constraints, i.e. what we called "generalized" DTPs and were discussed at Section 4.10.1. Staab is concerned notions of path-consistency in GNC but does not provide any experimental results. He claims that "*[GNCs]* are strictly more expressive than the networks permitted by Stergiou and Koubarakis" but without giving an example. Since DTPs with conjunctions in the disjuncts can be converted to simple DTPs, this claim is difficult to understand (also see Section 4.10.1).

Barber's TNC are also as expressive as DTP. A TNC is a TCSP with the addition of what Barber calls I-L-Sets. I-L-Sets (Inconsistent-Label-Sets) are essentially no-goods: an I-L-Set looks has the form $\neg(c_{ij} \wedge \ldots \wedge c_{mn})$ denoting that the conjunction does not hold in the TCSP. A constraint $a \vee b$, where a and b are STP-like constraints of the form $x - y \leq b_1$ and $w - z \leq b_2$ (involving two pairs of different variables) cannot be represented as a TCSP constraint, but it can be encoded as the I-L-Set $\neg(\neg a \wedge \neg b)$ (using De'Morgan's rule for Boolean Algebra). However, it is required that in a TCN the disjuncts in an I-L-Sets have to participate to some other TCSP constraint. Thus, it is not enough to just add the above I-L-Set, we also have to add TCSP constraints so that a and b appear in the underlying TCSP. For example, we can add the TCSP constraints $(a \vee \neg a)$ and $(b \vee \neg b)$. (Notice that if a is the constraint $5 \leq y - x \leq 10$, then $(a \vee \neg a)$ has to be represented as the TCSP constraint $-\infty \leq y - x \leq 5 \vee 5 \leq y - x \leq 10 \vee 10 \leq y - x \leq \infty$. Also notice that the additional constraints do not change the set of solutions). By using this scheme, TNCs reach the expressiveness of the DTP, albeit in a funny way. To solve TCNs, Barber in [Barber 2000] provides a path-consistency algorithm that in essence calculates the full set of all no-goods, without giving any experimental results. TCNs also have features to handle conditions and alternative contexts that will concern us later in Chapter 6.

4.9.2 Scheduling algorithms

Above, we discussed formalisms with expressive power roughly equivalent to the DTP. Surprisingly, most the work just described does not provide guidance in efficient DTP solving. The work discussed in Section 4.7 does, as does the literature from one other area, that of Automated Scheduling.

In fact, we might ask why is DTP solving scheduling. The quick answer is that it actually is. For example, in [Cheng and Smith 1995; Cheng and Smith 1995] a similar approach to DTP solving, called Precedence Constraint Posting (PCP) is applied to typical scheduling problems such as the Job-Shop Scheduling (JSSP) and the Hoist Scheduling Problem with very encouraging results. Cheng and Smith used a formalism based on the PIA in [Meiri 1991] and employed domain-specific heuristics.

The difference between other scheduling approaches and DTP and PCP solving is that typically scheduling is formulated as a CSP with variables that are the start times of the events. In other words, instead of searching for a consistent component STP, scheduling algorithms search in the space of complete time assignments. On the other hand, the way we defined the meta-CSP search of our DTP solver and in PCP, we are searching in the space of decisions, i.e. which disjunct to satisfy.

The PCP and DTP approach, according to Cheng and Smith (page 5, [Cheng and Smith 1995]) is that "by deferring commitment of specific start times until they are forced by problem constraints, a larger set of possible extensions is retained, reducing the likelihood of arriving at an inconsistent state. Likewise, from a solution robustness perspective, precise start time decisions are delayed (if possible) until needed at execution time. Finally, formulation as an ordering problem provides a more convenient search space in which to operate. The basic insight underlying PCP is that the search benefits provided by look-ahead analysis of resource contention over time can be obtained through much simpler, local analysis of sequencing flexibility". To these comments, we would like to add that execution flexibility is very important for planning applications.

Stergiou and Koubarakis [Stergiou and Koubarakis 2000] applied their DTP solver on JSSP with somewhat disappointing results. Nevertheless, we have to consider that DTP is a much more general problem than JSSP, there are many optimized and specialized JSSP schedulers armed with domain specific heuristics, and the JSSP has been studied in the literature for a long time. In addition, the Stergiou and Koubarakis DTP solver was the first of its kind. It would be very interesting to apply Epilitis to various scheduling problems, extended with the domain specific heuristics described in [Cheng and Smith 1995].

We would also like to note that typically it is easy or even trivial to find a solution to scheduling problems while it is hard to find an optimal solution. On the other hand, at least on the random DTP problems we have tested, just finding one solution is inherently hard. Scheduling research focuses on finding specialized algorithms and heuristics for particular problems, while the DTP has been developed as a general framework that subsumes a number of problems for scheduling and temporal reasoning representing them in a uniform way. As a more general problem, it is expected for DTP solvers to perform worst than specialized schedulers.

Let us add that the no-good learning techniques that we developed in this chapter for DTPs could potentially be applied to other planning and scheduling problems and are not restricted to DTPs. In general, the two areas of scheduling and temporal reasoning overlap and there could be a very fruitful exchange of ideas between the two. Cheng and Smith's experiments indicate that the constraint satisfaction techniques typically used in temporal reasoning definitely have application in scheduling and we hope our techniques will find their way into the scheduling community too.

Finally, no-good learning has also been recently applied to SAT solving with impressive results [Silva and Sakallah 1996], while it is gaining ground in the constraint satisfaction community too [Richards 1998].

4.10 Discussion, Contributions, and Future Work

DTP is a class of constraint-based temporal reasoning problems that was recently invented and appeared in the literature for the first time only in 1998 [Stergiou and Koubarakis 1998]. Even

though similar problems have been discussed in the literature for a decade [Dechter, Meiri et al. 1991], the idiosyncrasies of DTPs certainly require more research in order to be able to design efficient solvers.

4.10.1 Considering multiple conjunctions at once

Recall from the definition of DTPs that the allowable constraints take the form $c_{i1} \vee \ldots \vee c_{in}$ where each c_{ij} is an STP-like constraint of the form $x - y \leq b$. However, it might be beneficial for computational reasons to allow the constraints c_{ij} to be conjunctions of constraints, i.e. to take the form $p_{ij1} \wedge \ldots \wedge p_{ijk}$ where each p_{ijm} is an STP-like constraint. Allowing this more general form of constraint does not increase the expressive power of DTPs because each disjunctive constraint of the more general form could be converted to CNF forming (exponentially many to the number of p_{ijm}) "normal" disjunctive DTP constraints. We will call the DTPs with the restriction that each c_{ij} is an STP-like constraint, "simple" DTP, and the ones that allow c_{ij} to be conjunctions of STP-like constraints, "generalized" DTPs (see [Staab 1998]).

Example 4-7: Consider the following TCSP with just two variables x and y and the following single constraint:

$$2 \leq x - y \leq 5 \ \vee \ 10 \leq x - y \leq 15 \ \vee \ 20 \leq x - y \leq 25$$

If x is the time we arrive to our destination and y the time we start, the constraint might arise from the fact that if we take our car to our destination it takes between 2 and 5 minutes to arrive, if we take the bus it takes between 10 and 15 minutes, while it takes between 20 and 25 minutes to walk. The constraint is not in "simple" DTP form because the disjuncts are not constraints of the form $x - y \leq d$ but instead of the form $l \leq x - y \leq u$, i.e. the conjunction $(x - y \leq u \wedge y - x \leq -l)$. Converting the problem to "simple" DTP we get the following eight constraints:

$$\begin{array}{l}
x - y \leq 5 \ \vee \ x - y \leq 15 \ \vee \ x - y \leq 25 \\
x - y \leq 5 \ \vee \ x - y \leq 15 \ \vee \ y - x \leq -20 \\
x - y \leq 5 \ \vee \ y - x \leq -10 \ \vee \ x - y \leq 25 \\
x - y \leq 5 \ \vee \ y - x \leq -10 \ \vee \ y - x \leq -20 \\
y - x \leq -2 \ \vee \ x - y \leq 15 \ \vee \ x - y \leq 25 \\
y - x \leq -2 \ \vee \ x - y \leq 15 \ \vee \ y - x \leq -20 \\
y - x \leq -2 \ \vee \ y - x \leq -10 \ \vee \ x - y \leq 25 \\
y - x \leq -2 \ \vee \ y - x \leq -10 \ \vee \ y - x \leq -20
\end{array}$$

It is not clear whether converting a "generalized" DTP to a "simple" DTP would be computationally easier than directly solving the "generalized" DTP. To solve the "generalized" DTP directly, forward check and propagate should need to be made to handle conjunctions of constraints instead of a single STP-like constraint.. Intuitively, is it easy to see that the answer of which approach will give the best performance will depend on the number of conjunctions in the disjuncts: translating constraints with a large number of conjuncts will create an exponential number of "simple" DTP constraints. Allowing conjunctions in the disjuncts is not of purely academic interest because, as we will discuss, to determine if a Conditional Simple Temporal

Problem (see Chapter 6) is Dynamically Consistent we convert it to an appropriate "generalized" DTP problem. The DTP constraints that arise in this translation inherently contain a large number of conjunctions.

Below we present a list of issues to be considered in designing a solver for the "generalized" DTP problem:

- Forward-checking a conjunction $p_{ij1} \wedge \ldots \wedge p_{ijk}$ differs from forward-checking a single conjunct p_{ijm}. Theorem 4-2 does not hold which means that it is not as simple as checking the FC-condition. Probably the most efficient way to forward check the conjunction is to use the algorithm in [Brusoni, Console et al. 1997] [Brusoni, Console et al. 1997] which takes $O(k^3)$ time in the number of time-points involved in the conjunction. Alternatively, one could propagate all the constraints in the current STP, check for consistency, and then retract from the current STP to forward check the next value. A similar approach would be needed for the removal of the subsumed variables.
- When a DTP-solver maintains the distance array of the current STP forward checking a value takes constant time. Since however forward checking the $p_{ij1} \wedge \ldots \wedge p_{ijk}$ takes significantly more time, techniques for reducing the number of consistency checks, such as the incremental forward checking of Oddi and Cesta, will become more important.
- For the same reason, preprocessing techniques that remove as many values as possible from the variable domains will become increasingly important as the size of the conjunctions in the disjuncts increases and we predict that for "generalized" DTPs it will be a primary issue.
- Semantic branching will become difficult to implement: the negation of the conjunct $p_{ij1} \wedge \ldots \wedge p_{ijk}$ is $\neg p_{ij1} \vee \ldots \vee \neg p_{ijk}$ and we cannot propagate a disjunction in the current STP. However, the semantic branching constraint could be added to the current DTP as a form of no-good. Of course, whether this will result in increased performance must be examined empirically.
- No-good recording will become more important because the cost of rediscovering failures is greater. For example, to discover the no-good $\{C_1 \leftarrow c_{11}, C_2 \leftarrow c_{21}, C_4 \leftarrow c_{41}\}$ we performed two STP constraint propagations (of constraints c_{11} and c_{21}) and one forward checking operation on c_{41}. In a "generalized" DTP, where c_{11}, c_{21}, and c_{41} might contain a large number of STP-like constraints, the temporal propagation is considerably greater and the same no-good will save much more computational effort.

4.10.2 Contributions

Here follows a list of the most important contributions of this dissertation to DTP solving:

- The DTP was defined and presented and basic solving techniques based on search were presented. We surveyed the literature, identified all pruning techniques previously used and all methods for performing forward checking, classified them and analyzed them theoretically. We also analyzed and theoretically compared all current DTP solvers and identified the trade-offs each one favors.

- We designed new implementations that enable the simultaneous use of the different pruning techniques. This required first identifying and resolving their negative interactions.
- We conducted experimental investigations of all known pruning techniques and compared their relative strength and the synergies among them.
- We theoretically compared the CSP versus the SAT approach in DTP solving.
- We introduced a new pruning technique to DTP solving, namely no-good recording (learning), by adopting it from the CSP literature, and showed that it significantly improves performance. We experimented with no-good recording and found the optimal or a near optimal no-good recording size for the type of problems we used.
- We experimented with various heuristics, some of which use as heuristics information the no-goods recorded, and identified the best one.
- We provided experimental and theoretical evidence why FC-off in DTP solvers does not necessarily increase performance.
- We identified potential problems with using forward checks (also called consistency checks) as a machine-independent measure of performance, which is currently the adopted measure in the literature, and counter suggested a more accurate measure that also takes into consideration the constraint propagations.
- Combining all the previous pruning methods together, no-good learning with optimally bounded no-good size, the theoretically best way known so far of performing forward-checking (i.e. maintaining full path consistency and using the FC-condition), and the best from our heuristics, we built a DTP solver, called Epilitis, that improved run-time performance about two orders of magnitude over the previous state-of-the-art DTP solver.
- Epilitis can be used as a test-bed for experimenting with pruning techniques and various heuristics since it allows for the techniques to be turned off and on and is publicly available to encourage further experimentation by the community.

The applicability of this work has already been proven. For some time, an early version of Epilitis has been a core component of the a couple of real planning projects we have been involved with and has proven extremely effective. More specifically these projects are the Plan Management Agent, an intelligent calendar that manages the user's plans, and Nursebot, a robotic assistant intended to help elderly people with their daily activities. The latest results presented here are currently being incorporated in both projects.

4.10.3 Future Work

There are a great number of potential future research topics that we intend to explore and to which we encourage other researchers to engage in. Here, we provide but a small list:

- No-goods can be used for dynamic DTPs, i.e. sequences of DTPs that differ from successive elements by a few constraints. As we discussed in Chapter 3, it is possible that some of the no-goods identified and recorded for the previous DTP, still hold for the next DTP in the sequence, and thus can prune the search space. In one extreme case, if the no-good $<\varnothing, J>$ still holds in the next DTP, then the DTP is inconsistent and this is proven without any search performed! In [Schiex and Verfaillie 1994] it is shown that the method greatly improves the performance of CSP solvers on dynamic CSPs.
- We have but scratched the surface of efficient DTP solving. There are a great number of potential improvements to the solver, such as using other no-good algorithms [Richards 1998], random restarts [Gomez, Selman et al. 1998], better heuristics, and preprocessing techniques. We also expect a number of scheduling techniques such as profiling, to port to DTP solving.
- Even though DTPs are very expressive subsuming a large number of other temporal and scheduling problems, they are still not expressive enough for several planning applications. First, it is greatly desirable to combine features from DTPs and STPUs, i.e. disjunctions and uncertainty of timing of uncontrollable events. Second, as we further explore in Chapter 7, in order to use DTPs for conflict resolution in partial order planning, we need to be able to pose codesignation and non-codesignation constraints of the form $x = y$ and $x \neq y$ respectively, where x and y are not temporal variables, but resource variables instead. We defer a complete discussion of this subject, and in general the applications of DTPs in planning until Chapter 7.
- Further experimentation is required to investigate the performance of Epilitis, and in general the techniques we introduced, to scheduling problems, TCSP problems, other temporal reasoning problems, and DTP problems that exhibit structural patterns. The experiments are also necessary to generalize our hypotheses from randomly generated problems to DTPs that arise in practice.

5. FLEXIBLE DISPATCH OF TEMPORAL PLANS ENCODED AS DTPS

This chapter deals with the problem of dispatching temporal plans represented as DTPs while retaining all the execution flexibility. *The main result of this chapter is a dispatching algorithm for DTPs with several desired properties.*

5.1 Introduction

Many systems are designed to perform both planning and execution: they include a plan deliberation component to produce plans that are then dispatched to an execution component, or ***executive***, which is responsible for the performance of the actions in the plan. When the plans have temporal constraints, dispatch may be non-trivial, and the system may include a distinct ***dispatcher***, which is responsible for ensuring that all temporal constraints are satisfied by the executive. The nature and difficulty of the dispatch problem depends upon the temporal representation used in the plans. When all actions have fixed times, plan dispatch is straightforward: the dispatcher just submits the fixed schedule to the executive, which then has to perform the actions according to that schedule. However, plans with fixed schedules are brittle; to allow for the possibility of unanticipated events, it is often desirable to use plans with looser temporal constraints.

Consider a very simple plan with two actions, *x* and *y*, and temporal constraints specifying that y must begin sometime between 10:00 and 10:30 a.m. and no sooner than 15 minutes after the completion of another action *x*, which will itself must start sometime after 9:00 a.m. and will take 10 minutes to execute. Dispatching this plan requires propagating these constraints, to further determine the restrictions on when the actions can be executed. Some of the propagation can be done before execution begins: for instance, the initial constraints alone are sufficient to imply that *x* must begin no later than 10:05 a.m. Additional propagation is required during plan execution: once *x* is executed, then the earliest starting time for *y* can be determined.

The plan above could be represented as a Simple Temporal Problem (STP) [Dechter, Meiri et al. 1991]. Prior work on dispatch [Muscettola, Morris et al. 1998; Tsamardinos 1998; Tsamardinos, Morris et al. 1998; Wallace and Freuder 2000] has focused on this class of plans. In this chapter, we describe a dispatch algorithm that is applicable to a much broader set of plans, namely those that can be cast as Disjunctive Temporal Problems (DTPs). DTPs extend the expressiveness of STPs by allowing disjunctive constraints: we can represent, for example, that action *x* must begin either between 9:00 and 10:00a.m. *or* after 5:00p.m. In fact, DTPs are also strictly more expressive than Temporal Constraint Satisfaction Problem (TCSPs) [Dechter, Meiri et al. 1991], because they allow constraints across multiple variables. Thus, in a DTP we can represent the constraint that

either action *x* must be completed by noon or action *y* will take more than 2 hours. The increased expressiveness of DTPs makes it a suitable model for many planning and scheduling problems. Moreover, in the previous chapter quite efficient algorithms for solving DTPs were developed.

The role of a dispatcher is to notify the executive of when actions may be executed and when they must be executed. Informally, we will say that a dispatch algorithm is ***correct*** if, whenever the executive executes actions according to the dispatch notifications, the performance of those actions respects the temporal constraints of the underlying plan. Obviously, dispatch algorithms should be correct, but correctness is not enough: dispatchers should also be ***deadlock-free***: they should provide enough information so that the executive does not violate a constraint through inaction. A third desirable property for dispatchers is ***maximal flexibility***: they should not issue a notification that unnecessarily eliminates a possible execution, i.e., an execution that respects the constraints of the underlying plan. Finally, we will require dispatch algorithms to be ***useful***, in the sense that they really do some work. Usefulness will be defined as producing outputs that require only polynomial-time reasoning on the part of the executive. Without a requirement of usefulness, one could achieve the other three properties by designing a DTP dispatcher that simply passed the DTP representation of a plan on to the executive, letting it do all the reasoning about when to execute actions.

5.2 A Dispatch Example

This section presents an example of dispatching a temporal plan encoded as a DTP. To model a plan as an DTP, we define two nodes in V for each action Q in the plan: one node to denote the event of Q starting, and the other to denote its ending. This allows us to represent constraints on the duration of actions, as well as on the time intervals between the starts or ends of distinct actions. By defining a distinguished time-reference node ***TR***, we can also specify clock times of execution for actions. DTPs also allow the constraints to be disjunctive so that we can express the constraint that Q has a duration of either [2, 3] *or* [5,6] time units, or that Q has to be either before *or* after another action *R*.
We repeat the following DTP definitions as a reminder to the reader:

Definition 4-3: An ***exact solution*** to a DTP is an assignment to the DTP variables that respects all the constraints. A DTP is ***consistent*** if it has at least one exact solution otherwise it is called ***inconsistent***.

Definition 4-4: A ***consistent component*** *S* of the DTP *D* is also called a(n inexact) ***solution*** of *D*.

To illustrate the desired behavior of a dispatcher for DTP-based plans, we present a very simple example of a plan that has three actions, *P, Q,* and *R*. We suppose that the time intervals [5,10] and [15,20] after *TR* are the only allowable times for actions *P* and *Q*, and that *P* and *Q* interact negatively with one another if executed too closely in time, and so must be separated by at least 6 time units. We further suppose that action *R* must be performed either at [11,12] or [21,22] after *TR*. Finally, to simplify the presentation of the example, we assume that all three actions are instantaneous, and thus can be represented with a single DTP time-point. The plan as described can be represented with the following constraints:

$$C_1.\ 5 \le P - TR \le 10 \ \vee \ 15 \le P - TR \le 20$$
$$C_2.\ 5 \le Q - TR \le 10 \ \vee \ 15 \le Q - TR \le 20$$
$$C_3.\ 6 \le P - Q \le \infty \ \vee \ 6 \le Q - P \le \infty$$
$$C_4.\ 11 \le R - TR \le 12 \vee \ 21 \le R - TR \le 22$$

This DTP[19] has four solutions, i.e., four consistent component STPs:

1. STP_1: c_{11}, c_{22}, c_{32}, c_{41} (P in [5, 10], Q in [15, 20], R in [11, 12])
2. STP_2: c_{11}, c_{22}, c_{32}, c_{42} (P in [5, 10], Q in [15, 20], R in [21, 22])
3. STP_3: c_{12}, c_{21}, c_{31}, c_{41} (P in [15, 20], Q in [5, 10], R in [11, 12])
4. STP_4: c_{12}, c_{21}, c_{31}, c_{42} (P in [15, 20], Q in [5, 10], R in [21, 22])

As explained above, a component STP consists of one disjunct from each constraint of the DTP. For example, STP_1 includes the first disjunct of C_1 ($5 \le P - TR \le 10$), the second disjunct of C_2 ($15 \le Q - TR \le 20$), the second disjunct of C_3 ($6 \le Q - P \le \infty$) and the first disjunct of C_4 ($11 \le R - TR \le 12$). Each solution provides information about when events may and must be executed. An action *may* be executed if two conditions hold: it is *enabled*, and it is *live*.

Definition 5-1: An STP time-point x is ***enabled*** if and only if all the events that are constrained to occur before it have already been executed. A DTP time-point x is ***enabled*** if and only if it has a consistent component STP in which x is enabled.

An action is enabled when its start time is enabled. In STP_1, both P and R are initially enabled (at time TR=0). Q is not initially enabled, because of constraints c_{32}. However, Q is initially enabled in STP_3 and STP_4. Hence all three of the actions are initially enabled for the entire DTP.

Obviously, if an action is not enabled it should not yet be executed. Equally obviously, enablement is not a sufficient condition for execution: an action must also be ***live,*** in the sense that the temporal constraints pertaining to its clock time of execution are satisfied. In the current example, none of the actions are initially live: the first actions to become live are P and Q, at time 5. In general, one cannot directly determine the period of time during which an action is live by simply looking at the explicitly specified constraints: interactions amongst those constraints must also be considered. In an STP, one can determine when an action is live by computing the *distance graph*, which is an all-pairs shortest-path matrix for the nodes in the STP. An event (time-point) x is live exactly during the time interval $[-d_{xTR}, d_{TRx}]$, that is, the interval with lower bound equal to the negation of the minimal distance from x to TR, and upper bound equal to the minimal distance from TR to x. We will call that interval the time window for x.

Definition 5-2:: The ***time window of an STP time-point x*** is the interval $[-d_{xTR}, d_{TRx}]$. Given a set of consistent component STPs for a DTP, we will write TW(x,i) to denote the time window for variable x in the i^{th} such STP. The ***upper bound*** of a time window $[l,u]$ for x in STP i, written U(x,i), is u. The ***time window of a DTP time-point x*** is TW (x)=$\cup_{i \in S}$ TW (x,i), where S is the solution set of D.

Notice that the time-window of an STP time-point x is always a convex interval while the time-window of a DTP time-point can be non-convex.

---

[19] The DTP does not match Definition 4-1 for DTPs since it for example $5 \le P - TR \le 10$ is not in the form $x - y \le b$. Nevertheless, it is a "generalized" DTP (see Section 4.10.1) and it could be converted to a DTP. We do not convert it however for clarity of presentation of the example.

The dispatcher can provide information about when actions are enabled and live in an ***Execution Table (ET).*** This is a list of ordered pairs, one for each enabled action. The first element of the entry specifies the action, and the second is a list of the convex intervals in that element's time window. For our example, then, the initial ET would be:

$$\langle P, \{[5,10], [15,20]\}\rangle$$
$$\langle Q, \{[5,10],[15,20]\}\rangle$$
$$\langle R, \{[11,12],[21,22]\}\rangle$$

The ET summarizes the information in the solution STPs so that the executive does not have to handle them directly.

The ET provides information about what actions *may* be performed. Unfortunately, it does not provide enough information for the executive to determine what actions *must* be performed. To see this, note that the ET just given does not indicate that there is a problem with deferring both P and Q until after time 10. However, such a decision would lead to failure: as a result of the interactions of constraints C_1, C_2, and C_3, one of actions P or Q must be done by time 10. Analysis of the solution STPs reveals this fact: executing P after time 10 is inconsistent with STP_1 and STP_2, while executing Q after time 10 is inconsistent with STP_3 and STP_4. If the clock time reaches 11 and neither P nor Q has been executed, then all four solutions to the DTP will have been eliminated. Thus, in addition to the information in the ET, the dispatcher must also provide a second type of information to the executive. The ***deadline formula (DF)*** provides the executive with information about the next deadline that must be met.

Below we explain how to calculate the DF, which is more complicated than computing the ET, and we prove that a dispatcher that uses our algorithm has the desirable features of being correct, deadlock-free, maximally flexible, and useful under reasonable assumptions. Here we simply complete the example, by illustrating how the ET and the DF would be updated as time passes. The initial DF would indicate that either P or Q must be executed by time 10. Suppose that when at time 8, action P is executed. At this point, STP_3 and STP_4 these are no longer solutions to the DTP (more correctly, to the DTP augmented with the constraint P=8). The ET then becomes

$$\langle Q, \{[15,20]\}\rangle$$
$$\langle R, \{[11,12], [21,22]\}\rangle$$

and the DF is trivially (Q by 20). In this case, an update to ET and DF resulted because an activity occurred. However, updates may also be required when an activity does not occur within an allowable time window. For example, if R has still not been executed at time 13, then its entry in the ET should be updated to be just the singleton [21,22], with no changes required to the DF.

The example presented in this section contains variables with very little interaction. In general, there can be significantly more interaction amongst the temporal constraints, and the DF can be arbitrarily complex.

5.3 The Dispatch Algorithm

We now present our algorithm for the dispatch of plans encoded as DTPs. The input to algorithm is a DTP and an indication of the reason the dispatcher was invoked (explained below). The output is an Execution Table (ET) and a Deadline Formula (DF). As indicated above,

- an Execution Table (***ET***) is a list of pairs <x, TW(x)>, where x is an event in the DTP, and TW(x) is a time window for it, and
- a Deadline Formula (**DF**) is a pair <F, t>, where F is a CNF formula with only positive literals from the set of variables in the DTP being dispatched, and t is a time.

The semantics of the ET is that for each pair <x, TW(x)>, x has to be executed some time within TW(x). It is up to the executive to decide exactly when. The DF imposes the constraint that F has to hold by time t, where a variable that appears in the DF becomes true when its corresponding event is executed. The executive must execute events in such a way that F will hold by time t.

The dispatch algorithm will be called in three circumstances:

- A new plan needs to have its dispatch information initialized, at or before time TR.
- An event in the DTP is assigned an execution time. This occurs whenever an event is actually executed.[20]
- An opportunity for execution passes. This occurs whenever the clock time passes the upper bound of a convex interval in the time window for an action that has not yet been executed.

Initial-Dispatch (DTP D, STP[I] Solutions)
1. Find all *n* solutions (consistent component STPs) to D, calculate their *distance graphs*, and store them in Solutions[i]. Associate each solution with its (integer-valued) index.
2. Set the variable TR to have the status *Executed*, and assign *TR=0*.
3. *Output-ET*(Solutions)
4. *Output-DF*(Solutions)

Figure 5-1: Inital-Dispatch

Slightly different processing must be done in each case, as indicated in Figure 5-2, which gives the top-level dispatch algorithm. We will describe each case in turn, beginning with initialization; see Figure 5-1. Upon invocation, the solution set for the DTP is initially computed. After each solution is found, its *distance graph* is computed and stored away in the Solutions array. Finally, the ET and DF can be computed and output.

Dispatch (DTP D, STP [I] Solutions; Trigger T)
1. Case T:
2. T = Initialization: **Initial-Dispatch**(D, Solutions)
3. T = Event-Time-Assignment: **Update-for-Executed-Event**(Solutions);
4. T= Interval-Passed: **Update-for-Violated-Bounds**(Solutions);

Figure 5-2: Top-Level Dispatch Algorithm

Generating the ET is relatively straightforward; see Figure 5-3. For each DTP variable x we have to determine if x is enabled and therefore should be included in the ET. If it should, then we

[20] In general, we may also want to handle the situation in which the constraints on a future DTP event are tightened prior to the time of execution. In this paper, we omit this complication for presentational clarity.

Output-DF (STP [i] Solutions)

1. Let U = the set of upper bounds on time windows, U(x,i) for each still unexecuted action x and each STP i.
2. Let *NC*, the next critical time point, be the value of the minimum upper bound in U.
3. Let U_{MIN} = {U(x, i)| U(x,i) = *NC*}.
4. For each x such that U(x,i) ∈ U_{MIN}, let S_x = {i | U(x,i) ∈ U_{MIN}}
5. Initialize F = true;
6. For each minimal solution *MinCover* of the set-cover problem let $F = F \wedge (\vee_{x \mid Sx \in MinCover} x)$.
7. Output DF = <F, NC>.

Figure 5-4: Output-DF

have to calculate TW (x). To check if a variable is enabled in an STP, we check that every action that is constrained to come before x has already been executed. We can read the relevant actions y off the distance graph, as demonstrated below in Lemma 1. The time window for each such y can also be read off the distance graph, since it is precisely the interval [$-d_{xTR}$, d_{TRx}], as noted in Section 5.2.

Lemma 5-1: An action y is constrained to come before x in some STP i if and only if $d_{xy} < 0$, where d_{xy} is the minimal distance from x to y (recorded in the distance graph).

Proof.(<=) $y - x \leq d_{xy}$, since d_{xy} is the minimal distance between x and y. Then since $d_{xy} < 0$, $y < x$. (=>) Suppose to the contrary that y is constrained to come before x and $d_{xy} \geq 0$. Then, there is an execution in which $y - x = d_{xy} \geq 0$, which implies that $y \geq x$.

Output-ET (STP [i] Solutions)

1. For each event x in Solutions
2. If x is enabled
3. ET = ET ∪ <x, TW(x)>

Figure 5-3: Output-ET

Now we proceed with the calculation of the DF, which is more complicated and more interesting that the ET. To get a feel for how we compute the DF, recall the example in Section 5.2. Initially, at time *TR,* the DTP has four solutions. To determine the initial DF, we consider the next critical moment *NC,* which is the next time at which any action must be performed. This time is equal to the minimal value of all the upper bounds on time windows for actions, i.e., it is min{U(x, i)| x is an action in the DTP, and i is a solution STP}. For instance, in our example DTP, U(P, 1) = U(P, 2) = 10. The actions that may need to be executed by *NC* are those x such that U(x, i) = *NC* for some STP i. We create a list U_{MIN} containing ordered pairs <x, i> such that U(x, i) = *NC*. In our current example, U_{MIN} = {<P, 1>, <P, 2>, <Q, 3>, <Q, 4>}.

Now we perform the interesting part of the computation. If $<x, i>$ is in U_{MIN}, it means that unless x is executed by time NC, STP_i will cease to be a solution for the DTP. It is acceptable for STP_i to be eliminated from the solution set only if there is at least one alternative STP that is not simultaneously eliminated. This is exactly what the deadline formula ensures: that at the next critical moment, the entire set of solutions will not be simultaneously eliminated. We thus use a minimal-set cover algorithm [Cormen, Leiserson et al. 1990] to compute all sets of pairs $<x, i>$ in U_{MIN} such that the i values form a minimal cover of the set of solution STPs. In our example, there is only one minimal cover, namely the entire set U_{MIN}. Thus, the initial DF specifies that P or Q must be executed by time 10: $<P \vee Q, 10>$. In general, there may be multiple minimal covers of

Update-for-Violated-Bounds (STP[i] Solutions)
1. Let $U = \{U(x, k) \mid U(x, k) < \textit{Current-Time}\}$
2. Remove from *STP Solutions* all STPs k that appear in U.
3. **Output-ET** (Solutions).
4. **Output-DF** (Solutions).

Figure 5-5: Update-for-Violated-Bound

Update-for-Executed-Event (STP [i] Solutions)
1. Let x be the event that was executed at time t.
2. Remove from Solutions all STPs i for which $t \notin TW(x,i)$.
3. Propagate the constraint $t \leq x - TR \leq t$ in all remaining *STP Solutions*.
4. Mark x as *Executed*.
5. **Output-ET** (Solutions).
6. **Output-DF** (Solutions).

Figure 5-6: Update-for-Executed-Event

the solutions STPs: in that case, each cover specifies a disjunction of actions that must be performed by the next critical time. For instance, suppose that some DTP has four solution STPs, and that at time TR, $U(L, 1) = U(L, 2) = U(M, 3) = U(M, 4) = U(N, 4) = U(S, 3) = 10$. Then by time 10 either L or M must be executed; additionally, at least one of L or N or S must be executed. The corresponding DF is $<(L \vee M) \wedge (L \vee N \vee S), 10>$. The algorithm for computing and outputting the DF is given in Figure 5-4.

So far, we have described the processing that should be performed when the dispatcher initially creates the ET and DF. As mentioned, these structures must be updated whenever some upper bound(s) are violated or an event occurs. Figure 5-5 shows the former case: the main addition there is the removal of STPs that have ceased to be solutions to the DTP because the passage of time has invalidated one of their constraints. Figure 5-6 shows the latter case: there, after eliminating invalidated STPs, it is also necessary to propagate the new constraint through the remaining STPs in the solution set

5.4 Formal Properties of the Algorithm

We now prove that the algorithm presented in the previous section has the four desirable properties for dispatchers identified in Section 5.1. To do this, we first need to be more precise about how the dispatcher and the executive interact. The dispatcher issues a ***notification sequence***, a list of pairs $<ET,DF>_1 \ldots ,<ET,DF>_n$, with a new notification issued every time an event is executed or an upper bound is passed. (Thus, the i^{th} notification does *not* necessarily occur at time i.). The executive performs an ***execution sequence***, a list $x_1 = t_1, \ldots, x_n = t_n$ indicating that event x_i is executed at time t_i, subject to the restriction that $j>i \Rightarrow t_j > t_i$. An execution sequence is complete if it includes an assignment for each event in the original DTP; otherwise it is partial.

In practice, the notification sequence and the execution sequence will be interleaved in an ***event sequence***. Every time an action is performed, our algorithm updates the ET and DF, and thus issues a new notification. Thus there will never be two consecutive executions in the event sequence, but there may be two consecutive notifications, when a new notification results from an upper-bound violation. A typical event sequence thus has the following form:

$$<ET,DF>_1, x_i=t_i, <ET,DF>_2, <ET,DF>_3, x_j=t_j, \ldots$$

We can associate each execution event with the preceding notification, e.g., in this example, the notification for x_i, $Notif(x_i) = <ET,DF>_1$.

Definition 5-3: An execution sequence E ***respects*** a notification sequence N *iff*

- For each execution event $x_i=t_i$ in E, $<x_i, TW(x_i)>$ appears in ET of $Notif(x_i)$ and $t_i \in TW(x_i)$. That is, each event is performed in its allowable time window.
- For each $DF=<F,t>$ in N, $\{x_i \mid x_i = t_i \in E \text{ and } t_i \leq t\}$ satisfies F by time t. That is, the execution sequence satisfies all the deadline formulae.

In our proofs below, we need to make one assumption, namely, that both the dispatcher and executive are arbitrarily fast, and in particular, the time taken by the functions **Update-for-Violated-Bound** and **Update-for-Executed-Event**, as well as the time it takes for the executive to decide what and when to execute, is negligible. This assumption means that the dispatch algorithm is always up-to-date: it never "misses" a requirement to eliminate a candidate STP from the set of solutions to the original DTP. Later we will discuss ways of weakening this assumption.

We now prove that the dispatch algorithm presented in the previous section is correct, deadlock free, maximally flexible, and useful.

Definition 5-4:: Given a DTP D, a dispatcher is ***correct*** if any complete execution sequence that respects its notifications also satisfies the constraints of D.

Theorem 5-1: The dispatcher described by the function **Dispatch** is correct.

Proof: Consider an arbitrary execution event $x_i=t_i$ in the execution sequence. Because the execution sequence respects the notifications, $<x_i, TW(x_i)>$ appears in ET of Notif (x_i) and $t_i \in TW(x_i)$. This means that $t_i \in TW(x,m)$, for some STP m that is a solution to D at time t_i. Moreover, m must be consistent with all the previous execution events; otherwise the **Dispatch** function—in particular, **Update-for-Executed-Event** would have

removed m from the solution set. Thus, if the execution sequence is complete, it is an exact solution of some STP m that is a solution of the original DTP. ♦

Definition 5-5: Given a DTP D, a dispatcher is ***deadlock-free*** if any partial execution that respects its notifications can be extended to a complete execution that satisfies the constraints of D.

Theorem 5-2: The dispatcher described by the function **Dispatch** is deadlock-free.

Proof: It is sufficient to show that **Dispatch** never empties the solution set until after a complete execution sequence. **Dispatch** removes STPs from the solution set in two circumstances: after an activity is executed and after an upper bound is passed. In the former case, **Dispatch** retains at least one STP, namely the one consistent with the execution sequence so far (m in the proof of Theorem 5-1). We thus consider the latter case, in which an upper bound $U(x,i)$ has passed. **Update-for-Violated-Bounds** will remove invalidated solution STPs, including i. However, this cannot result in the removal of all STPs, because if it did, i.e., if some action needed to be performed by $U(x,i)$, then **Output-DF** would already have produced a DF warning, and since the execution sequence respects the dispatch notifications, the relevant action would already have been performed. ♦

Definition 5-6: Given a DTP D, a dispatcher is ***maximally flexible*** if every complete execution sequence that respects the constraints in D will be part of some complete event sequence.

Theorem 5-3: The dispatcher described by the function **Dispatch** is maximally flexible.

Proof: Every execution sequence S: $x_1=t_1, \ldots, x_n=t_n$ that satisfies the constraints of D will be a solution to some consistent component STP m of D. It suffices to show when the executive follows S, m will not be removed from the set of STP solutions. **Update-for-Executed-Event** will not remove m because m is consistent with S, and **Update-for-Violated-Bounds** will not remove m for the same reason (i.e., every $t_i \in \mathrm{TW}(x_i,m)$.) ♦

Definition 5-7: A dispatcher is ***useful*** if generating an execution sequence in polynomial in the size of the notifications.

Theorem 5-4: The dispatcher described by the function **Dispatch** is useful.

Proof: We describe an algorithm that is polynomial in n, where n is the size of the ET and DF:

1. Find a variable x that is in both DF and ET such that the current clock time $CT \in \mathrm{TW}(x)$ and execute $x=CT$.
2. If there is no such variable, pick any other variable y in ET such that $CT \in \mathrm{TW}(y)$ and execute $y=CT$.
3. Otherwise, wait.

The algorithm is obviously polynomial in the size of the ET and DF. What is left to prove is that this algorithm respects the notification sequences. We first prove a lemma.

Lemma 5-2: If a variable x appears in some DF $<F, t>$, it becomes ready for execution at or before time t (i.e. $t \in$ TW (x)).

Proof: If x is in DF, then its upper bound $U(x,m)$ is t for some STP m, by the way we construct the DF. Therefore, TW(x, m) has upper bound t and so $t \in$TW(x) is satisfied and thus x is live at t (and possibly before, since t is an upper bound). In addition, for all variables y that are constrained to appear before x in all STP solutions, it is the case that U (y, j) < U (x, j). But, since by the construction of the DF, U (x, j) is the minimum upper bound of all yet-to-be-executed variables, y must have been executed and so x is enabled at (and possibly before) t.

Proof of main theorem: Now we are ready to prove that an executive that uses the algorithm above respects the dispatch notifications. First, it is easy to see that each event is executed in its time window (i.e. the first requirement of Definition 5-3 is satisfied). We need to show that it is also the case that if the dispatcher outputs DF=$<F, t>$ at time t', then F will be satisfied by time t by an executive using the algorithm above (second requirement in Definition 5-3). Suppose F is not satisfied by time t. Then there is at least one clause in F that is false, which means that all variables in that clause are false, which in turn means that there is some event x in that clause which has not yet been executed. By Lemma 2, x would have been ready by time t. If x belongs to any F' of any subsequent DF=$<F', t'>$ it will be picked for execution by step 1 of our algorithm just above. Otherwise, x will eventually be picked up for execution by step 2 before time t. Therefore, at least one of $x \in X$ would have been executed by the executive which contracts the assumption that F is false at time t. Since DF=$<F, t>$ is always true by time t, the variables executed until time t satisfy the formula F, i.e . $\{x_i \mid x_i = t_i \in E$ and $t_i \leq t\}$ satisfies F as the definition requires. ♦

Our proofs relied on the assumption that the amount of time taken by the dispatcher and the executive are both negligible. Indeed, if in any solution STP two variables x and y are constrained to occur at the same instant, then only an infinitely fast dispatcher and executive can successfully process the event sequence $x=t_x$, $<ET, DF>_i$, $y=t_y$, where $<ET,DF>_i$ is the result of a call to **Update-for-Executed-Event**. There are, however, a few remedies to the unrealistic need for infinite speed.

Suppose that we determine that the dispatcher takes at most time e to response to any updates. If two variables x and y have to be executed within e then there is no time for the sequence of calls $x=t_x$, **EventExecuted**, $y=t_y$. Only if the call to **EventExecuted** could be dropped can x and y be executed. For each STP, the dispatcher could determine all sets S_l consisting of events that are constrained to occur too close together to enable it to run its updating functions. Then one approach would be to augment the output of the dispatcher with these sets S_L and to perform updating only at the end of execution of each set. Another approach would be to allow the dispatcher to work in two different modes. The normal mode is the one we described above, in which the dispatcher tracks in parallel the execution of a set of STPs. The dispatcher could be in this mode whenever there is enough time to run its update functions without missing any upper bounds. In the "emergency" mode, the dispatcher would arbitrarily pick one still valid STP solution and force the executive to follow that STP. Executing an STP can be done within real

time guarantees that can be predetermined [Tsamardinos 1998]. We defer details of both approaches to future work.

5.5 Discussion and Contributions

DTPs belong to a family of constraint-based formalisms for temporal reasoning. Dispatch has been previously studied for simpler members of this class, including STPs [Tsamardinos 1998], as well as for extensions that model resource usage [Wallace and Freuder 2000]. For STPs and TCSPs, the typical method of dispatch involves converting the original network to one that is *minimal*, in the sense that the domain of each variable contains only values that belong to some solution. Dispatch using a minimal network is straightforward: the executive can always choose to perform any enabled event x, provided that the clock time is within the set of values associated with that event (since all values in the minimal network belong to some solution). After an execution decision has been made (i.e., $x=t$ gets fixed), this constraint is propagated in the minimal network, the values that do not belong in any solution are removed, and the process repeats. Deadline formulae do not need to be explicitly calculated. An STP or TCSP can be converted to a minimal network by performing path consistency. A similar approach does not work for DTPs, however, because where STPs and TCSPs allow only binary constraints, DTPs allow n-ary

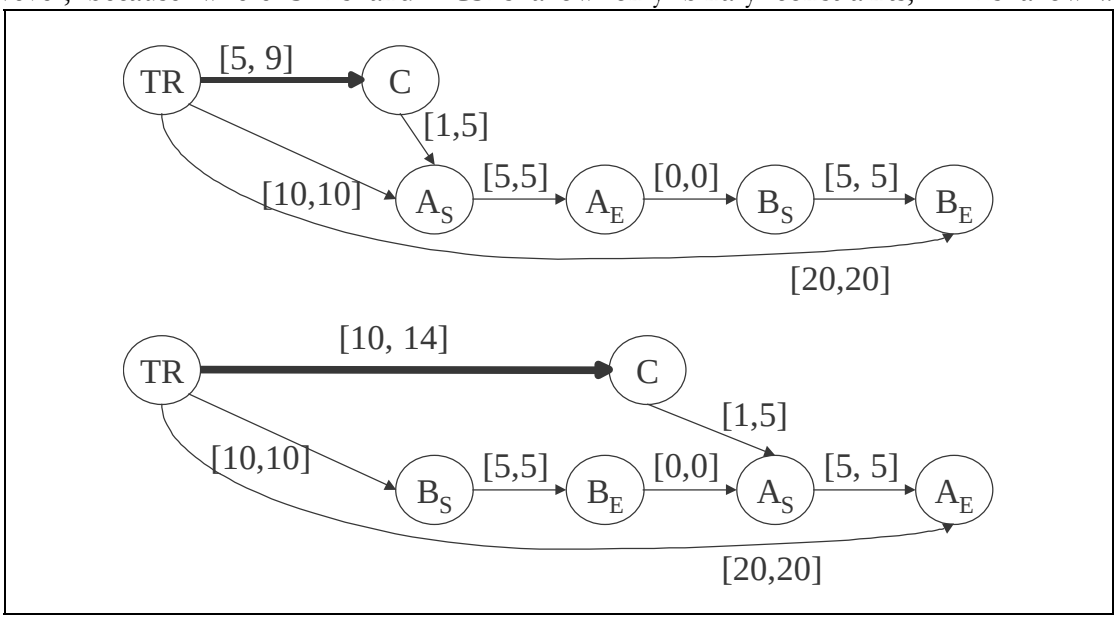


Figure 5-7: A temporal problem that cannot be correctly executed by an STPU, but it can if it is represented as a DTP (The uncontrollable link is indicated with boldface)

constraints, and it is thus not obvious what type of consistency algorithm can be used to determine the minimal network.

Recent work has considered a different extension to STPs, namely Simple Temporal Problems with Uncertainty (STPUs) [Vidal and Ghallab 1996; Coradeschi and Vidal 1998; Morris and Muscettola 1999; Morris and Muscettola 2000; Vidal 2000; Vidal 2000; Vidal and Morris 2001; Vidal and Coradeschi August 1999] (see also Section 2.5). STPUs are designed to encode problems

in which certain events are outside the control of the executive. Many real-world environments involve uncontrollable events, whose timing is determined by other agents or by "Nature". If formalisms such as STPs or DTPs are used for some domains, then dispatchers cannot ensure that temporal constraints will be met: at best, they can specify times for which the executive should perform controllable events to ensure *their* compliance with temporal constraints, hoping that the uncontrollable events occur during their time windows. In contrast, STPUs explicitly model uncontrollable events, and research has led to techniques for determining different levels of controllability in a network. For example, an STPU is *dynamically controllable* if it can be executed in such a way that the temporal constraints will all be satisfied regardless of the timing of the uncontrollable events. It turns out, however, that there are cases in which uncontrollability can better be modeled with DTPs than with STPUs.

To see this, consider an example with two tasks, *A* and *B*, each of which takes exactly five time units, and which are constrained not to overlap. Assume that *A* must start precisely at time 10, and *B* must end precisely at time 20. Assume further that there is an uncontrollable event *C* that will occur within the window [5, 14], and that *A* is constrained to start at least one and at most five time units after *C* occurs. The scenario can be represented with a DTP with two solution STPs showed in Figure 5-7. In the top one, *C* occurs before time 9, and the executive must perform task *A* and then task *B*. In the lower one, *C* occurs after 10, and the executive must perform *B* before *A*. Both of these networks are dynamically controllable. In contrast, the alternative, of representing the problem with a single STPU, would necessitate a particular order of tasks *A* and *B*, which, when combined with the complete interval on the uncontrollable link, is not dynamically controllable.

The DTP framework does not model uncontrollable events explicitly, and thus cannot properly account for the possible shrinkage of uncontrollable events during execution. On the other hand, the STPU framework does not allow for disjunctive choice, and as the example above shows, this omission may lead to certain problems that are dynamically controllable being misclassified. Clearly, it would be desirable to have a framework that allows both disjunctive constraints and the explicit representation of uncontrollable events; this is an important topic for future research.

Finally, the list below summarizes the **contributions** in this chapter:

- The problem of DTP dispatching was presented and the desired properties for a dispatcher were defined, i.e. being ***correct, deadlock-free, maximally flexible,*** and ***useful***..
- A dispatching algorithm that provably solves the problem with the desired properties was presented.

6. TEMPORAL REASONING WITH CONDITIONAL EVENTS

This chapter deals with the definition of new temporal reasoning frameworks that can reason with quantitative temporal constraints and conditional execution contexts. The new formalisms defined are called Conditional Simple Temporal Problem (**CSTP**) and Conditional Disjunctive Temporal Problem (**CDTP**). The main result of this chapter is consistency-checking algorithms for the different notions of consistency in CSTP and CDTP. We prove their correctness and complexity and identify tractable subclasses.

6.1 Introduction and Related Research

A number of systems and applications need to be able to reason with alternative contexts, situations, intentions, trends, plans, possible worlds or causal projections and to know what holds in each one of them [Dousson, Gaborit et al. 1993; Gerevini and Schubert 1995; Srivastava and Kambhampati 1999]. In addition, these systems may have to impose temporal constraints on events and actions belonging to different contexts. This section defines a novel class of temporal problems, called Conditional Simple Temporal Problems (***CSTP***) and its disjunctive counterpart Conditional Disjunctive Temporal Problems (***CDTP***) that are able to represent alternative contexts and temporal constraints. In addition we provide a theoretical analysis of CSTPs and CDTPs, identify three different notions of consistency, and provide reasoning algorithms for determining each kind of consistency. The formalization is based on STPs and DTPs respectively, which are then extended to include conditions that define the alternative contexts. CSTPs were first introduced in [Tsamardinos, Pollack et al. 2000] but without a formal definition or appropriate reasoning algorithms given.

There is ample evidence that many systems need to reason simultaneously about time and different contexts and situations, partly displayed by the intense research in branching and partial order time (an overview of branching time systems is at [Penczek 1991]). The branching time approach is logic based and adopts a tree-structured view of time; every time instant may have several immediate successors that correspond to different futures. In a partial order approach the situation is similar except that every time instant may also have several immediate predecessors corresponding to different pasts. Branching time logics, like other temporal logics [Orgun and Ma 1994], typically contain modal operators such as □ – to denote a formula always holds at every state, o - to denote that a formula holds at the next state, and ◊ - to denote that a formula eventually holds. There are several reasons to develop logics based on branching time or partial order structures. For example, such logics can be used for model checking of programs: for a given program and given input, an execution tree is generated by the program. Over the execution,

universal properties involving all computations can be studied as well as existential properties referring to a specific computation. This approach is very useful for non-deterministic programs and can also be applied for concurrent programs [Clarke and Emerson 1981; Pnueli 1981].

Reasoning about different possible futures and contexts, typically the result of uncertainty in effects of actions, exogenous events, or partial observability of the current and initial state, has been given special consideration in planning too. At the one extreme there are systems that consider domains in which there is a great deal of uncertainty from all the factors mentioned above. These systems use Markov Decision Processes [Boutilier, Dean et al. 1999; Jonsson, Haslum et al. 2000] or Universal Planning techniques [Schoppers 1987; Jonsson, Haslum et al. 2000]. These systems discover a policy, i.e. a complete plan that specifies for every possible initial state and every possible action outcome the optimal action to take next. The other extreme of course is classical planning where there is no uncertainty, there is only one context, and there is a single known initial state.

In between the two ends, there is conditional (or contingency) planning [Peot and Smith 1992; Pryor and Collins 1993; Pryor and Collins 1996], where the state of the world is only partially known and thus special observation actions are inserted to decide which course of action to follow next. A different course of action is created for every set of states for which the current plan is not applicable. In probabilistic planning [Draper, Hanks et al. 1994; Kushmerick, Hanks et al. 1995] actions have non-deterministic outcomes and when an uncertain outcome threatens the achievement of a goal the planner makes contingency plans for all (or only the most important [Onder 1999]) possible outcomes. The two approaches have been unified in [Draper, Hanks et al. 1994; Onder, Pollack et al. 1998; Onder and Pollack 1999]. In conformant planning [Smith and Weld 1998; Cimatti and Roveri 1999] a sequence of actions is constructed that achieves the planning goal for any possible initial state and non-deterministic behavior of the planning domain. In some sense, conformant planning can be viewed as reasoning about all possible pasts of the system.

As in conditional planning, CSTPs and CDTPs also encode special observation (or decision) nodes that (partially) determine the current execution context. However, none of the current planning approaches mentioned above (conditional, probabilistic, and conformant planning) can represent quantitative temporal constraints; instead they are typically restricted to ordering constraints. The same is true of the temporal logics. Additionally, to the best of our knowledge, all other approaches that are able to encode quantitative temporal constraints are unable to reason with different contexts. CSTPs and CDTPs permit the representation of and reasoning about quantitative constraints in alternative contexts.

A notable exception is described in [Barber 2000] (also described in Chapter 4), which introduces a temporal representation scheme, called Temporal Constraint Network (TCN). TCNs are based on TCSPs allowing a limited form of disjunctions among the time-points. Barber introduces the concept of I-L-Sets (named after the terms Inconsistent Labeled Sets; essentially no-good assignments that declare a set of disjuncts inconsistent with each other) with which TCNs reach the expressive power of DTPs. However, in our opinion this way of representing n-ary temporal constraints is difficult to use, counter-intuitive to understand, and possibly computationally harder to solve (see Chapter 4). Barber also describes an extension to this scheme that can handle different contexts and is discussed in more detail at Section 6.9.

Another class of temporal reasoning problems that is related to CSTPs is the Simple Temporal Problems with Uncertainty (STPU) (presented at Chapter 2) [Vidal and Ghallab 1996; Vidal 2000]. In STPUs, there is uncertainty in the exact timing of events. So for example, in addition to the normal STP-like constraints of the form $l \leq x - y \leq u$, where the planning agent can assign any times to x and y as long as they respect the constraints, there are contingent constraints $l \leq w - z \leq u$, where it is Nature (or some other uncontrollable agent) that assigns time to w within the bounds $[l+z, u+z]$ (see Chapter 2 for more details). Similarly to the notions of CSTP consistency that we will define, i.e. Strong, Weak, and Dynamic Consistency, there is Strong, Weak, and Dynamic Controllability for STPU. The notions are analogous but unfortunately the theoretical results from the STPU analysis do not directly apply to CSTPs.

STPUs, CSTPs, and CDTPs all deal with uncertainty but with different forms of it. When viewed as representing plans, CSTPs and CDTPs represent uncertainty in terms of the courses of action that the agent that executes the temporal plan has to take, depending on the outcome of observations. Thus, the planning agent can assign any time it desires (respecting the constraints) to the time-points of the CSTP but it does not control the outcome of observations. In STPUs, there are no observations and no different contexts, but the agent does not control the timing of certain events (time-points). Obviously, it would be highly desirable to combine the features of both classes of problems to be able to reason with both sources of uncertainty. We now proceed by first presenting CSTPs and then extend them to CDTPs.

6.2 Definitions

This section provides some necessary notation, and then formally defines CSTPs and the different kinds of CSTP consistency. We will denote propositions with capital letters usually from the beginning of the alphabet, e.g. *A, B, C*. We will denote literals as A, or $\neg A$, and conjunctions of literals as lists of literals, e.g. $AB\neg C$ to denote $A \wedge B \wedge \neg C$.

Definition 6-1: A ***label*** l is a conjunction of literals. E.g. $AB\neg C$

Definition 6-2: Two labels l_1 , l_2 are ***inconsistent*** (denoted as **Inc(l_1 , l_2)**) if $l_1 \wedge l_2 \Rightarrow$ *False* . E.g. if $l_1 = AB$ and $l_2 = \neg B \neg C$.

Definition 6-3: Two labels l_1 , l_2 are ***consistent*** (denoted as **Con(l_1 , l_2)**) if $\neg$ Inc(l_1 , l_2).

Definition 6-4: A label l_1 ***subsumes*** label l_2, (also called l_1 ***is more specific*** than l_2), (denoted as **Sub(l_1 , l_2)**) if $l_1 \Rightarrow l_2$. E.g. $l_1 = AB\neg C$ and $l_2 = A\neg C$

Definition 6-5: The set of all possible labels defined with respect to a set of propositions P, is the ***label universe*** of P, $\boldsymbol{P^*}$.

We are now ready to formally define CSTPs.

Definition 6-6: A **Conditional Simple Temporal Problem (CSTP)** C is a tuple $<V, E, L, OV, O, P>$, where

P is a finite set of timed propositions {A, B, C, …} (e.g. "*rain* at time t")
V is a set of nodes (variables, time-points, events)

E is a set of constraints (edges) between the nodes of the form $l \leq x - y \leq u$, where l (u) is called the lower (upper) bound of the edge

L is a function $V \rightarrow P^*$ attaching a label to each node, e.g. L(n) = AB

$OV \subseteq V$ is the set of observation nodes

O is a 1-1 and "onto" function $P \rightarrow OV$ associating an observation variable with a proposition. *Thus, O(A) denotes the node that provides the truth-value for proposition A.*

The semantics of a CSTP are as follows. The time-points in V correspond to instantaneous events that can be assigned time (i.e. be executed). The assigned times need to respect the constraints in E. A node *v need only to be executed if its label becomes True.* At the time of their execution the observation nodes provide the truth-value of propositions, which in turn determine the truth-value of labels. If there are two observations for the same "physical" proposition A, but at different observation nodes, these observations should correspond to different CSTP propositions, e.g. A_{T1} , A_{T2} : *each CSTP proposition represents the truth-value of a proposition at the time of its observation.* Notice that the main difference from an STP is that nodes have a label assigned and they are only executed if their label becomes true. It is reasonable to assume that in any well-defined CSTP there should not be any edges between nodes with inconsistent labels: no constraint should hold between them since they will never be executed under the same circumstances. Also, it is reasonable to require that the constraints of the CSTP are such that if a proposition A is part of a label l of a node, both the node and the observation for A will be always be executed together, and that the node is constrained to occur after O(A). Otherwise, it is possible that when the node is to be executed, the truth-value of A will still be unknown. To ensure this requirement for any node U for which A appears in $L(U)$., we can statically check that *Sub(L(U), L(O(A))* and add the constraints O(A) < U to the CSTP.

Example 6-1: A CSTP is shown at Figure 6-1 where V = {*TR, X, Y, Z, U, W*}. As with STPs, we will sometimes use a special *time reference point TR*, associated with a specific arbitrary point in time, to be able to encode absolute constraints such as that X should be executed on Monday 1/1/2002, 12am. All non-annotated edges in the figure are assumed to have bounds [0, +∞]. The labels of the nodes are $L(TR) = L(X) = L(Y) = L(Z) = True$, $L(W) = A$, and $L(U) = \neg A$. There should be exactly one observation variable that provides the truth-value of A and it can be any of the nodes *TR, X, Y,* and *Z*. The CSTP of the example could represent a conditional plan (in which the actions are assumed instantaneous for clarity of presentation) in which actions *X, Y, Z,* and either *W* or *U* are executed.

Definition 6-7: An ***execution scenario*** s is a label that partitions the nodes in V into two sets V_1 and V_2 : V_1 = {$n \in V$: Sub(s , L(n))} and V_2 = {$n \in V$, Inc(s , L(n))}, i.e. the set of nodes that will be executed because l implies their label is *True* , and the set of nodes that will not be executed because s implies their label is *False*. An execution scenario contains all information to decide which nodes to execute and which not to execute. We will call the set of scenarios ***Sc***.

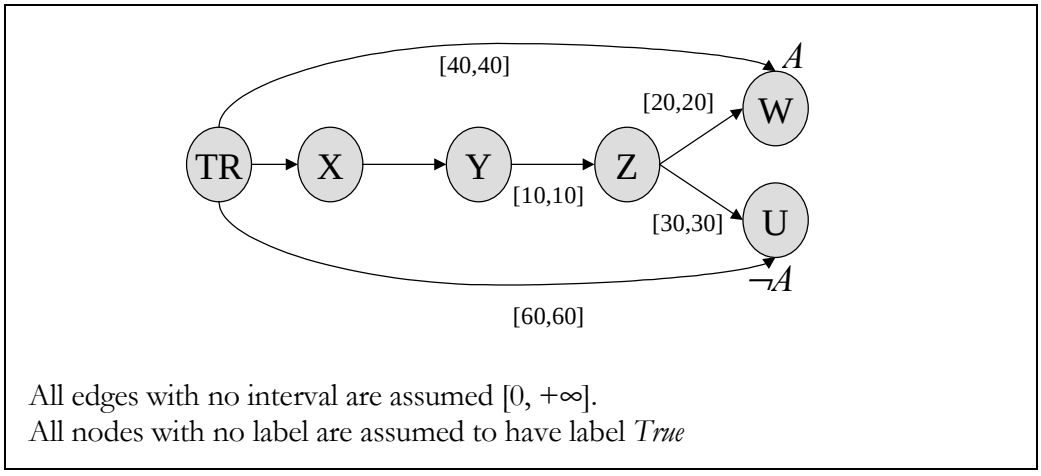


Figure 6-1 An example CSTP

Theorem 6-1: Any complete assignment to the propositions in P is a scenario (we will call it a ***complete scenario***).

Proof: Let s be a complete assignment, l a label and $p_1 \ldots p_n$ be the propositions that appear in l (either positive or negated). The set of p_i will also appear in s since s is complete. If any p_i appears with different sign in s and l then s and l are inconsistent. Otherwise, if all p_i appear with the same sign in s and l, then s subsumes l. So, every node, no matter what its label is, will either belong to V_1 or V_2 in Definition 6-7.

Definition 6-8: A ***projected STP*** of the CSTP $<V, L, E, OV, O, P>$ ***of an execution scenario s*** **,** denoted as ***Pr(s)*** is the STP $<V_1, E_1>$, where $V_1 = \{n \in V : \text{Sub}(s, \text{L}(n))\}$ and $E_1 = \{(v_1, v_2) \in E \mid v_1, v_2 \in V_1\}$. Put simply, the projected STP of an execution scenario s is the nodes of the CSTP that will be executed under s and all the edges among them.

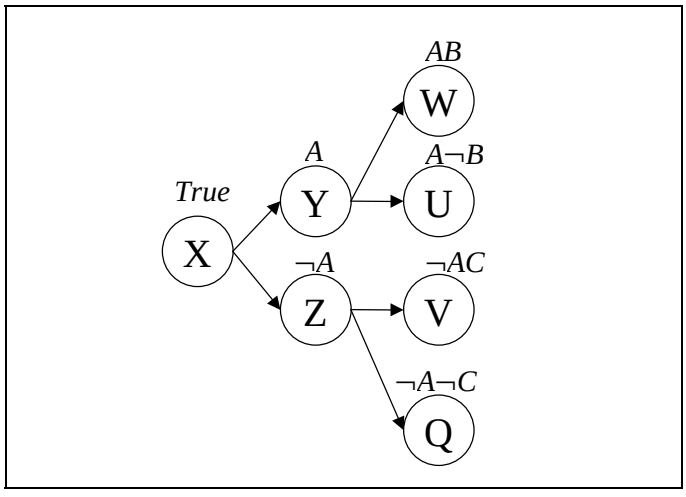


Figure 6-2: Another example CSTP

The number of all possible execution scenarios is at least $2^{|P|}$ and at most $3^{|P|}$. To see this notice that by Theorem 6-1 all the $2^{|P|}$ complete assignments are scenarios. Since every proposition can be either absent, positive, or negative in a label, there are $3^{|P|}$ labels, and it is possible that any of them partitions the nodes and is an execution scenario[21].

Now consider a CSTP with nodes {*X, Y, Z, W, U, V, Q*} with labels taken respectively from the following set: {*True, A*, ¬*A, AB*, *A*¬*B*, ¬*AC*, ¬*A*¬*C*} (i.e. *L(X)=A*) as in Figure 6-2 . Let us also suppose that *X* observes *A, Y* observes *B*, and *Z* observes *C*. The CSTP has the typical structure of the output of a conditional plan which first executes *X* (no matter what), observes *A,* then it executes *Y* if turns out *True* and so on. The execution scenarios *AB*, *ABC* and *AB*¬*C* all refer to the same execution, i.e. they partition the nodes to the same V_1 and V_2 sets in Definition 6-7: under all these three scenarios *X, Y,* and *W* will be executed and no other nodes.

Definition 6-9: Two execution scenarios are ***equivalent execution scenarios*** if they induce the same partition on the set of nodes.

The "equivalent execution scenarios" relation induces an equivalence class relation ***R***. A class of ***R*** contains all scenarios that are equivalent.

Definition 6-10: A scenario is a ***minimal execution scenario*** if it contains the minimal number of propositions in its "equivalent execution scenario" class. It is easy to see the minimal scenario is unique for a given class.

The number of minimal execution scenarios is at most $2^{|P|}$. In the example discussion above, labels *ABC*, *AB*¬*C,* and *AB* all belong to the same equivalent class, with *AB* being the minimal execution scenario of the class. For the rest of this chapter, *the term scenario will be referring to a minimal execution scenario*, unless otherwise stated. For the example above, there are four (minimal) scenarios: {*AB, A*¬*B,* ¬*AC,* ¬*A*¬*C*}.

6.2.1 Consistency in CSTPs

CSTPs have a different notion of consistency than TCSPs, DTPs, STPs, and most other temporal reasoning class of problems. In CSTPs, consistency is not defined simply as the existence of an assignment to the temporal variables (time-points) that respects the constraints. Assigning times to the variables without any other restrictions does not take into concern the uncertainty (i.e. the observations) represented in the network. When executing a CSTP, depending on how the information about the observations is pouring into the execution algorithm, there are three notions of consistency that we have identified so far.

First, it might be the case that the nodes have to be assigned times in advance, before execution begins, e.g. if the scheduling algorithm is not part of the execution system. In this case, the scheduling algorithm has no information about the outcome of the observations and should schedule the nodes in such a way that the constraints are respected no matter how the observations turn out. If such a schedule exists, then we will call the CSTP Strongly Consistent.

As a second case consider the following scenario. An agent has to plan now for a later point in time without knowing with certainty the exact future initial state. So, it has to plan for every

[21] Actually, it is easy to prove that if all propositions in *P* appear in at least one label, then there are at most $2^{|P|}$ labels.

possible initial state, even though for a particular initial state the plan might be deterministic with no observations and branches. As a special case, the truth value of all propositions on the labels are made just before execution to determine the initial state and thus, each scenario corresponds to a specific initial state, which will actually be executed will be known as soon as execution begins. In this case, what matters is whether there is a schedule that respects the constraints for each possible scenario. If this is possible we will call the CSTP Weakly Consistent.

These two special cases are certainly interesting both theoretically and for certain types of systems as described above. However, we expect the most typical and interesting case to be when the outcome of the observations is provided during execution and the schedule can be adjusted dynamically. In this case, an agent would like to know whether it is possible to execute the CSTP in such a way that no matter what the outcome of the observations turn out to be, it can keep extending the current partial CSTP solution (i.e. the partial execution of the time-points) so that it respects the temporal constraints. If this is possible then the CSTP is Dynamically Consistent.

We now provide a few more definitions leading to the formal definitions of these three notions of consistency. We then continue with specific examples to illustrate the concepts.

Definition 6-11: A ***schedule T*** of a CSTP $<V, E, L, OV, O, P>$ is a mapping $V \rightarrow \Re$, i.e. a time assignment to the nodes in V. We denote with T(v) the time assigned to node v. The same T that schedules the nodes in V can also be considered a time assignment for all projections $Pr(s)=<V', E'>$ if we define that the times assigned to the nodes in $V-V'$ by schedule T, always respect the constraints E' by default.

Definition 6-12: A CSTP $<V, E, L, OV, O, P>$ is ***Strongly Consistent*** if there is schedule T, such that for every $s \in \boldsymbol{S}$, T is a solution to ***Pr(s)***.

Definition 6-13: A CSTP $<V, E, L, OV, O, P>$ is ***Weakly Consistent*** if for every scenario $s \in \boldsymbol{Sc}$, there is schedule T, such that T is a solution to ***Pr(s)*** .

Definition 6-14: Consider two schedules T_1 , T_2 for scenarios s_1 , s_2 respectively. T_1 and T_2 totally order the observation nodes (and all nodes) in ***Pr***(s_1) and ***Pr***(s_2) into the observation sequences $seq_1 = n_{11}$, …, n_{1k} and $seq_2 = n_{21}$, …, n_{2l} (also totally ordered with respect to each other). We will say that two observations n_{1p} and n_{2q} ***differ*** if they either observe a different proposition, or they observe the same proposition and the outcome is different in the two scenarios. We will call the ***distinguishing time of s_1 and s_2*** , relatively to T_1 and T_2 and denote it as $Dis(s_1, s_2, T_1, T_2)$, the time $T(n)$ of the first observation node n that differs in the two sequences seq_1 and seq_2 (i.e. before node n, all pairs n_{1n} and n_{2m} do not differ).

Example 6-2: The above definition is indeed complex, but behind it lies an intuitive concept. Consider the scenarios $s_1 = ABCD$, $s_2 = ABC\neg D$, and $s_3 = ACD$. Also suppose that T_1 , T_2 , and T_3 are three schedules for the three scenarios respectively that order the observation of these propositions in the same order they appear in the definition of the scenarios, e.g. T_1 schedules the observations for A, B, C, and D in this order. Then $Dis(s_1, s_2, T_1, T_2) = min(T(n_1), T(n_2))$ where n_1 and n_2 are the observation nodes for D in s_1 and s_2 respectively. This is because these are the first nodes that differ in the two sequences because their outcome is different. $Dis(s_1, s_3, T_1, T_3) = min(T(n_3), T(n_4))$ where n_3 and n_4

are the observation nodes for B and C in s_1 and s_3 respectively. Again, these are the first observation nodes that differ in the two sequences since they observe different propositions. Thus the distinguishing time of s_1 and s_2 , relatively to T_1 and T_2 is the time of the first time-point when we can distinguish between the two scenarios.

Definition 6-15: Let us assume a total ordering of the observation nodes *order* in all scenarios. Then, any two schedules T_1 and T_2 in Definition 6-14 consistent with *order* induce the same total order for seq_1 and seq_2 and thus $Dis(s_1, s_2, T_1, T_2)$ does not depend on the exact schedules T_1 and T_2 but just on the *order*. We will call the node of $Dis(s_1, s_2, T_1, T_2)$ for any two schedules consistent with *order*, the ***distinguishing observation node*** between s_1 , s_2 relatively to the total order *order* we assumed and denote it as $\boldsymbol{Dis(s_1, s_2, order)}$.

Definition 6-16: An ***execution strategy St*** is a mapping from the set of scenarios to a schedule $St : \boldsymbol{Sc} \rightarrow T$. A ***viable*** execution strategy is one which $St(s)$ is a solution to $Pr(s)$ for each scenario $s \in \boldsymbol{Sc}$.

Definition 6-17: A CSTP $<V, E, L, OV, O, P>$ is ***Dynamically Consistent*** if there is a viable execution strategy St such that, for every pair of scenarios $s_1 , s_2 \in \boldsymbol{Sc}$ and for every node $v \in V$, if $T_1(v) \leq Dis(s_1, s_2, T_1, T_2)$ or $T_2(v) \leq Dis(s_1, s_2, T_1, T_2)$, then $T_1(v) = T_2(v)$, where $T_1 = St(s_1)$ and $T_2 = St(s_2)$. We will call the St that satisfies the definition a ***successful execution strategy.*** In other words, there is a set of schedules for each scenario (that is another way of viewing an execution strategy) such that for every pair of scenarios, the corresponding schedules schedule the nodes that appear before the distinguishing time at the same time.

Let us consider a node n with label *True* that is constrained to be before all observation nodes. This node will have to be executed in all scenarios. Moreover, since it is constrained to appear before any observation nodes, it is before the distinguishing time of every pair of scenarios. So, for any pair of scenarios n has to be scheduled the same time, and thus, transitively, it has to be scheduled at the same time in all scenarios.

To compare the three notions of consistency, reconsider the CSTP of Figure 6-1 as defined in Example 6-1. Is the network Strongly Consistent? We can immediately see that W has to be scheduled at time 40 because of constraint $40 \leq W - TR \leq 40$. Similarly, U has to be scheduled at time 60. Because of constraints $20 \leq W - Z \leq 20$ and $30 \leq U - Z \leq 30$, Z has to be scheduled at time 20 if A is observed *True*, and at time 30 if A is observed *False*. Therefore, there is no way we can schedule Z without knowing the truth-value of A and so the CSTP is not Strongly Consistent.

However, it is Weakly Consistent. To prove this we just have to provide a consistent schedule for each scenario. In this example there are two scenarios $s_1 = A$ and $s_2 = \neg A$. For s_1 a consistent schedule is $T_1(TR) = 0$, $T_1(X) = 5$, $T_1(Y) = 10$, $T_1(Z) = 20$, $T_1(W) = 40$ and for s_2 , $T_2(TR) = 0$, $T_2(X) = 10$, $T_2(Y) = 20$, $T_2(Z) = 30$, $T_2(U) = 60$.

Is the network Dynamically Consistent? Above we showed that $T(Z)$ must be 20 if A is observed *True*, and $T(Z)$ must be 30 if A is observed to be *False*. Similarly we find that $T(Y)$ has to be 10 in s_1 and 20 in s_2 . Therefore, if we observe A before Y we can determine which scenario we fall into and schedule Y, Z, accordingly, and depending on the scenario, also schedule W or U.

Formally, let us assume the observation point for A is X (i.e. $O(A)=X$) and consider the schedules $T_1(TR) = 0$, $T_1(X) = 5$, $T_1(Y) = 10$, $T_1(Z) = 20$, $T_1(W) = 40$ and $T_2(TR) = 0$, $T_2(X) = 5$, $T_2(Y) = 20$, $T_2(Z) = 30$, $T_2(U) = 60$. The distinguishing point for s_1 and s_2 is $T_1(X) = T_2(X) = 10$ (and so the distinguishing observation is X) since X is the only observation node and it has different outcomes for the two scenarios. For the execution strategy St where $St(A)=T_1$ and $St(\neg A)=T_2$, for every pair of scenarios (there is only one in this example), T_1 and T_2 are solutions to the STP projections of s_1 and s_2 respectively (thus St is viable). Also, for every node that is scheduled before or at time 10 (i.e. for nodes TR and X) the scheduled time is the same in both T_1 and T_2 . Thus, the network is Dynamically Consistent.

Before we provide algorithms for determining consistency, among other theoretical results, let us state an obvious property of the three notions of consistency:

Theorem 6-2: Strong Consistency ⇒ Dynamic Consistency ⇒ Weak Consistency

6.1 Strong Consistency

We now present an important property of Strong Consistency in a CSTP.

Theorem 6-3: A CSTP $<V, L, E, OV, O, P>$ is Strongly Consistent if and only if the STP $<V, E>$ is consistent.

Proof: The theorem states that we can determine Strong Consistency by ignoring the label and the observation information of the nodes in the CSTP and just calculate consistency as we would for an STP.

"⇒" Suppose that the CSTP is Strongly Consistent. Then we will show that the STP $<V, E>$ is also consistent. Let T be the schedule that is the solution to all $\boldsymbol{Pr(s_i)} = <V_i, E_i>$ for every scenario s_i . Since T satisfies the constraints in every set E_i is satisfies the constraints in their union $\cup_i E_i = E$. Thus T is a solution to $<V, E>$ and so it is consistent.

"⇐" Suppose that the STP $<V, E>$ is consistent, we will prove that the CSTP $<V, E, L, OV, O, P>$ is consistent. Let T be any solution of $<V, E>$ (it has to have at least one since it is consistent). For every s_i , T is also a solution of $\boldsymbol{Pr(s_i)} = <V_i, E_i>$ since $E_i \subseteq E$. Thus, T is a solution to every $\boldsymbol{Pr(s_i)} = <V_i, E_i>$ as the definition requires. ♦

The implication of this theorem is that any STP consistency-checking algorithm [Chleq 1995; Cesta and Oddi 1996] can be used for Strong Consistency Checking in a CSTP.

6.2 Weak Consistency is co-NP-complete

Theorem 6-4: Checking Weak CSTP Consistency is co-NP-complete.

Proof: We will prove the result by translating in polynomial time and space a SAT problem to the co-problem of checking Weak Consistent, the co-problem being finding a scenario s_i such that $Pr(s_i)$ is inconsistent. Specifically, we will create a CSTP given a SAT problem such that the SAT problem has a solution, if and only if there is a scenario s_i such that $\boldsymbol{Pr}(s_i)$ is inconsistent.

Given the SAT problem with Boolean variables $B = \{x, \ldots, y\}$ and clauses of the form $C_i = (x \vee \ldots \vee y \vee \neg z \ldots \vee \neg w)$, $i=1...K$ we create a CSTP $<V, E, L, ON, O, P>$ as follows: The set of propositions is $P = B = \{x, \ldots, y\}$. For each clause $C_i = (x \vee \ldots \vee y \vee \neg z \ldots \vee \neg w)$ and each variable appearance x or $\neg x$ in C_i we create a time-point X that we include in V, with label $L(X) = x$ or $L(X) = \neg x$ respectively. Let us denote with $Clause(C_i)$ the nodes of the CSTP that were included because of C_i. Since we are checking for Weak Consistency it does not matter which nodes are observation nodes. The last thing to define is the edges between the nodes. There is an edge annotated with the interval [–1 , –1] between a variable $X \in Clause(C_i)$ to all other variables with consistent labels in $Clause(C_{(i+1) \bmod K})$ annotated with the interval [-1, -1] (we will drop the mod K clause in the

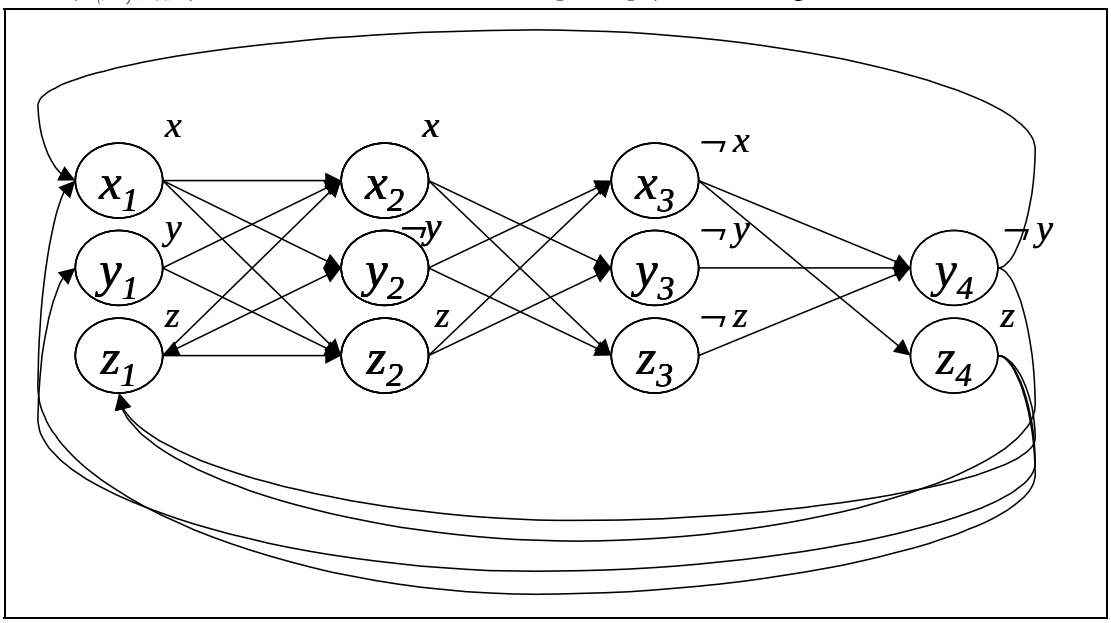


Figure 6-3: The CSTP resulting from translating a simple example SAT problem

rest of the proof for clarity; just remember that the nodes in the last $Clause(C_i)$ are connected to the nodes in the first $Clause(C_1)$). The translation is obviously linear in the number of SAT variables and linear in the number of clauses of the SAT problem.

A picture worth a thousand words, especially in this case, and so Figure 6-3 presents an example, by showing the resulting CSTP from the SAT problem $(x \vee y \vee z) \wedge (x \vee \neg y \vee z) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (\neg y \vee z)$. The labels of each node appear on its top right corner. Notice that there are three propositions in the CSTP for the three SAT variables x, y, z that appear in the labels, either as positive or negative literals, and eleven CSTP nodes one for each appearance of a variable in any clause. The nodes in the CSTP are arranged in columns corresponding to $Clause(C_i)$, $i=1\ldots4$ and are named with the variable of the corresponding proposition and the index i. The edges are connected from a node in $Clause(C_i)$ to all the nodes in $Clause(C_{i+1})$. The order of appearance of clauses in the SAT problem is arbitrary. Also recall that all the edges have a [–1, –1] interval attached with

them not shown in the figure for clarity. Notice also that there are no edges between nodes with inconsistent labels, e.g. x and $\neg x$.

Let us assume that the SAT problem has a solution {x=*True,* …, y=*True,* z=*False,*…, w=*False*}. Since this solution makes true at least one variable in a clause, it will make true the label of at least one CSTP time-point within *Clause*(C_i) . We will call a time-point whose label becomes true in *Clause*(C_i) L_i (there may be more than one). Notice that each L_i has to have an edge to L_{i+1} because it cannot be the case that L_i has a label x and L_{i+1} a label $\neg x$ by the way the SAT solution is constructed (it never assigned x both *True* and *False* at the same time). Thus the set {L_i , i=*1*…K} forms a negative cycle (with weight (-1)*K). There must be at least one scenario s that makes all the nodes in {L_i , i=*1*…K} true. Namely, let s be the complete scenario that corresponds to the SAT solution (s is a scenario by Theorem 6-1). In other words, ***Pr(s)*** is an inconsistent projected STP. In the above example, the SAT solution {x=*True,* y=*False,* z=*True* } makes the labels of the CSTP nodes x_1 , x_2 , y_2 , y_3 , y_4 , z_1 , z_2 ,and z_4 *True* and all the rest False. The former set forms at least one negative cycle, e.g. {x_1 , x_2 , y_3 , z_4 }. This completes the proof that if the SAT problem has a solution, the CSTP is not Weakly Consistent.

We will now prove the converse, namely that if the SAT problem has no solution, then the CSTP is Weakly Consistent. Take any complete assignment to the SAT variables. Any such assignment also corresponds to a scenario of the CSTP. If SAT has no solution, for every such assignment/complete scenario s there is at least one clause C_i that is not true, or in other words, all the SAT literals of C_i have to be false too. Thus, all the time-points of *Clause*(C_i) are inconsistent (not executed) under scenario s. But, by the way the CSTP is constructed, every cycle (negative or not) has to go through all clauses. Since no time-point in *Clause*(C_i) becomes true under s , there can be no negative cycle in ***Pr(s)*** for any scenario s.

The above argument shows that checking Weak CSTP Consistency is co-NP-hard. Since checking if an STP is consistent is a polynomial problem, co-Weak Consistency is also in co-NP and thus the problem is co-NP-complete.♦

6.3 Calculating the Set of Minimal Execution Scenarios

By Definition 6-13, in order to check Weak Consistency we can run an STP consistency-checking algorithm for each projected STP. This constitutes a brute force algorithm for the problem. To calculate a projected STP we need the scenario of which it is a projection. Two equivalent execution scenarios obviously have the same projected STPs and thus to speed up the brute force algorithm, it is more efficient if we only pick only one scenario per equivalence class of ***R*** (see the paragraph following Definition 6-9 for details on ***R***); in particular, we would like to identify from each class of ***R*** the minimal execution scenario, without having to generate all possible scenarios. This section deals with the problem of calculating the set of the minimal execution scenarios of a CSTP. We will call this the problem of ***calculating the set of minimal execution scenarios*** and provide exact and approximate algorithms to solve. An approximate algorithm might return a set of scenarios that are not minimal, but will nonetheless include at least

one representative for each class in ***R***. One trivial and bad approximation algorithm would be to return the set of all complete scenarios, i.e. scenarios that assign a truth-value to every proposition in *P*.

MakeScenarioTree(set of propositions left *P*, current label *L*, set of remaining labels to partition *Labels*)

1. *node* = **new-tree-node**(*L*); *node.LeftChild* = *node.RightChild* = *Nil*
2. If *L* does not subsume all labels in *Labels* Then
3. p = **choose-next-proposition**(*P*, *L*, *Labels*)
4. $Labels_{left}$ ={$l \in$ Labels |Con($L \wedge p$, l) $\wedge\neg$Sub(*L*, l)}, P_{left} = the propositions in $Labels_{left}$
5. *node.LeftChild* = **MakeScenarioTree**(P_{left}, L$\wedge p$, $Labels_{left}$)
6. $Labels_{right}$ = {$l \in$ Labels | Con($L\wedge\neg p$, l) $\wedge\neg$Sub(*L*, l)}, P_{right} = the propositions in $Labels_{right}$
7. *node.RightChild*=**MakeScenarioTree**(P_{right} , L$\wedge\neg p$, $Labels_{right}$ }
8. EndIf
9. Return *node*.

Figure 6-4: The algorithm for generating the execution scenario tree.

We now present an algorithm (Figure 6-4) that creates a set of execution scenarios covering all classes in ***R*** according to the heuristic function **choose-next-proposition**. Next we present a case where it is easy for the heuristic to make an optimal choice, but it is not applicable in every situation. The generalization of this case provides an optimal exact heuristic that guarantees an exact solution to calculating the set of minimal executions scenarios. Nevertheless, making the optimal choice is expensive and so an alternatively greedy heuristic is proposed.

The algorithm takes a set of labels as input and recursively splits them to those that are unclassified by proposition p (i.e. those who are consistent with p but not subsumed by it) and those that are unclassified by proposition $\neg p$, thus gradually building a tree. In the end, the tree leaves will only have labels consistent with the path of propositions from the root node. Thus, each leaf will correspond to a scenario.

Formally, the algorithm takes as input a set of propositions *P* that appear in the labels *Labels*, the current label *L* corresponding to the propositions already in the path from the root to the current node, and the set of remaining labels *Labels* to partition. The initial recursive call to the algorithm is the set of all the CSTP propositions, the label *True*, and the set of all the labels that annotate the CSTP nodes. The algorithm will generate a tree whose nodes are annotated with labels and whose leaf labels correspond to the approximation to the set of minimal execution scenarios. The call to **MakeScenarioTree** first creates a new tree node with label *L* (line 1). Thus, the first node created will always have label *True*. If all nodes have label *True* (i.e. the CSTP degenerates to an STP) this will be the only node created and returned. If this is not true, then a proposition p will be chosen at line 3 and the algorithm will be called recursively with at least one less proposition available P – {p}. The left child returns the scenario tree to the reduced problem of splitting the labels that are unclassified by L$\wedge p$ while the right child returns the scenario tree that splits the labels unclassified with L$\wedge\neg p$.

Example 6-3: Let us illustrate the algorithm with a trace on an example problem. Suppose we have a CSTP with nodes associated with labels from the set {*A, ¬A, AB, A¬B, ABC, AB¬C*}. This set of labels would have been produced for example by a conditional planner that first branches on observation for *A*, then if *A* is true, it branches on *B*, then if

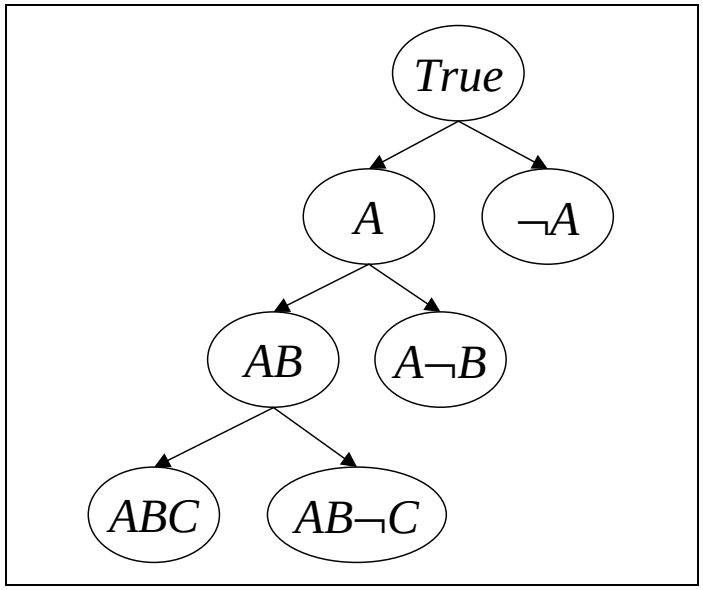


Figure 6-5: The tree generated by MakeScenarioTree on Example 6-3

B is true it branches on *C*. Assuming that this is the order function choose-next-proposition returns the variables (i.e. first *A,* then *B*, then *C*) the tree produced by **MakeScenarioTree** is shown in Figure 6-5.

The first call to the algorithm is with parameters *P*={*A, B, C*}, *L* = *True*, and *Labels* = {*A, ¬A, AB, A¬B, ABC, AB¬C*}. Since *True* does not classify all the labels in *Labels*, the algorithm is called recursively. The left child call is with parameters *P*={*B, C*}, *L* = *A*, and *Labels* = {*AB, A¬B, ABC, AB¬C*}. Continuing in this fashion, the next call to the left child will be with parameters *P*={*C*}, *L* = *AB*, and *Labels* = {*ABC, AB¬C*}, until the whole tree is constructed.

Notice that the leaves of the tree have to be scenarios, i.e. they split the nodes into a set of CSTP nodes that should be executed and to a set that should not be executed, because they split the set of all labels that appear in the CSTP: every leaf label either subsumes some label of the CSTP or is inconsistent with it. For example the label *ABC* is consistent with {*A, AB, ABC*} and inconsistent with all other labels.

Continuing with the example, consider the effect of the function **choose-next-proposition**. Let us suppose that instead of this function returning the propositions in the order *A, B, C* it returns them in order *A, C, B*. Then, the scenario tree produced is shown at Figure 6-7. In the worst case the tree produced will have $2^{|P|}$ leaves. In this example,

this will happen if **choose-next-variable** returns the propositions in the order *C, B, A* (Figure 6-6).

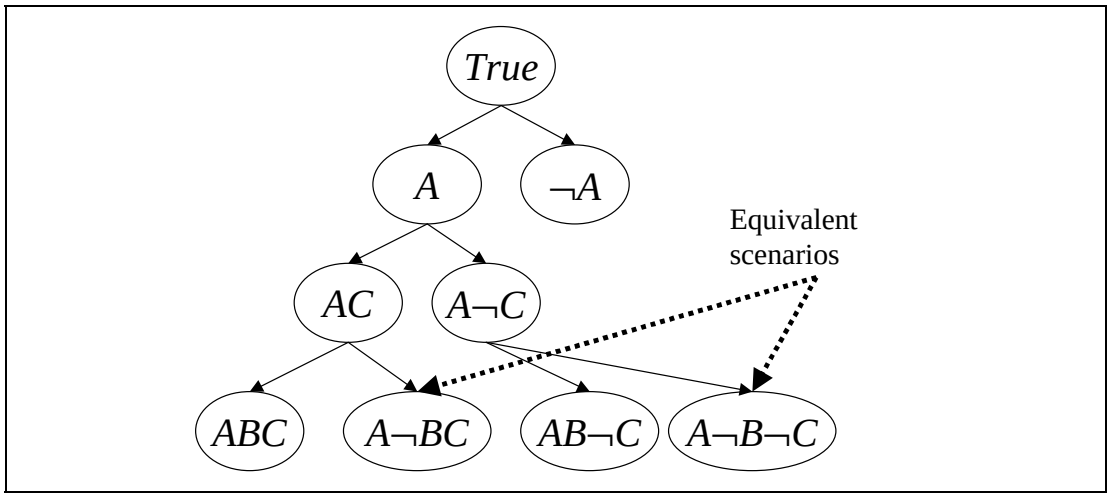


Figure 6-7: The tree generated by MakeScenarioTree on Example 6-3 with a different proposition ordering heuristic.

Next we consider the question of a good heuristic for implementing the function **choose-next-**

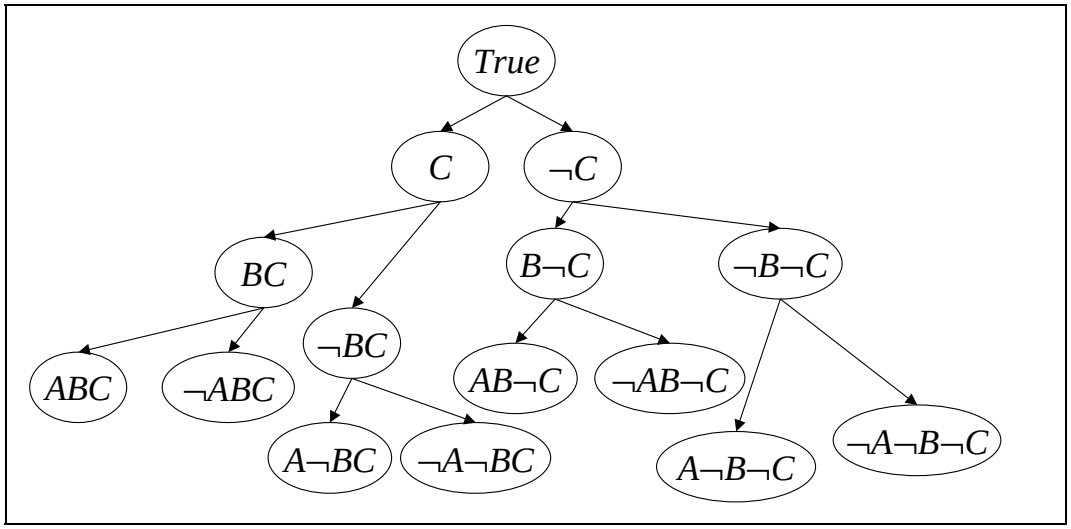


Figure 6-6: The tree generated by MakeScenarioTree on Example 6-3 for the worst case order of propositions.

proposition. In every node called with parameters *P, L,* and *Labels*, the proposition *p* selected can be considered to split the set *Unclassified* = $Labels_{left} \cup Labels_{right}$ (i.e. the set of labels passed to the children) to three mutually disjoint subsets:

- $\boldsymbol{S}_p = \{l \in Unclassified \mid S(l, L \wedge p)\}$
- $\boldsymbol{S}_{\neg p} = \{l \in Unclassified \mid S(l, L \wedge \neg p\}$
- $\boldsymbol{S}_{\varnothing p} = \{l \in Unclassified \mid \neg S(l, L \wedge p)\}$ and $\neg S(l, L \wedge \neg p\}\}$,

or in other words, the set S_p of all remaining labels that contain p, the set $S_{\neg p}$ that contain $\neg p$, and the set $S_{\varnothing_p}$ in which p does not appear at all. Notice that S_p , $S_{\neg p}$, and $S_{\varnothing_p}$ are pairwise disjoint and that $Unclassified = S_p \cup S_{\neg p} \cup S_{\varnothing_p}$.

Now, if we pick p as the next proposition in the algorithm, then the left child will be given to split the label set $S_p \cup S_{\varnothing_p}$ and the right child the label set $S_{\neg p} \cup S_{\varnothing_p}$. To see this notice that the set of labels that are consistent with $L \wedge p$ (line 4 in the algorithm) are exactly those in which either p appears positive or does not appear at all.

Going back to Example 6-3, let us calculate the sets S_p , $S_{\neg p}$, $S_{\varnothing_p}$ for the root node for each $p \in \{A, B, C\}$, current label $L = True$, and for the set of labels $\{A, \neg A, AB, A\neg B, ABC, AB\neg C\}$:

$S_A = \{A, AB, A\neg B, ABC, AB\neg C\}$	$S_{\neg A} = \{\neg A\}$	$S_{\varnothing_A} = \{\}$
$S_B = \{AB, ABC, AB\neg C\}$	$S_{\neg B} = \{A\neg B\}$	$S_{\varnothing_B} = \{A, \neg A\}$
$S_C = \{ABC\}$	$S_{\neg C} = \{AB\neg C\}$	$S_{\varnothing_C} = \{A, \neg A, AB, A\neg B\}$

Thus, if we chose to branch on proposition A, the left child will be given the set $Labels_{left} = S_A$ and the right child the set $Labels_{right} = S_{\neg A}$. If we choose to branch on B, the left child will be given the set $Labels_{left} = S_B \cup S_{\varnothing_B}$ and the right child the set $Labels_{right} = S_{\neg B} \cup S_{\varnothing_B}$, and if we choose to branch on C, $Labels_{left} = S_C \cup S_{\varnothing_C}$ and $Labels_{right} = S_{\neg C} \cup S_{\varnothing_C}$. It is easy to see intuitively, that the best choice is to initially branch is on A, and the worst choice is to branch on C.

We know formally prove the results of the above discussion, as a specific case of a more

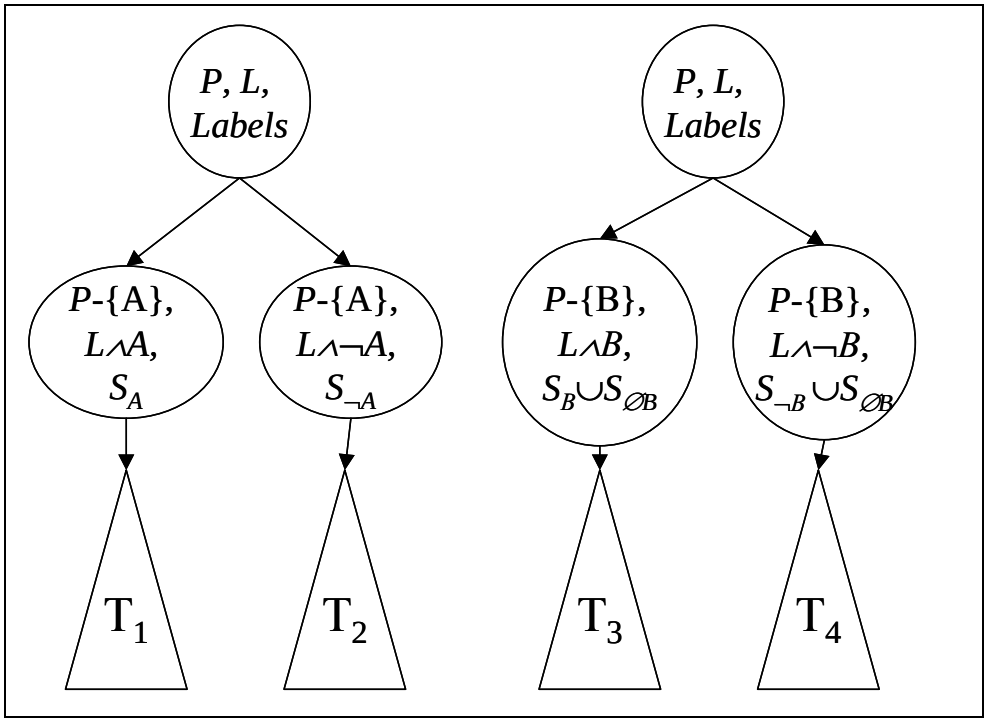


Figure 6-8: The trees *Tree$_A$(P, L, Labels)* and *Tree$_B$(P, L, Labels)* for the proof of Theorem 6-5.

general theorem that provides the proof for the optimal implementation of **choose-next-proposition** so to build the smallest execution scenario tree:

Theorem 6-5: Suppose **MakeScenarioTree** (Figure 6-4) is called with parameters *P, L, Labels*, and that there is a proposition $A \in P$, such that $S_{\varnothing_A} = \varnothing$ (relatively to the three parameters). Let us call $Tree_A(P, L, Labels)$ the tree returned by **MakeScenarioTree**(*P, L, Labels*) if **choose-next-proposition** returns *A*. Then *A* is an optimal choice for **choose-next-proposition**, i.e. $Tree_A(P, L, Labels)$ has the smallest number of leaves than any other $Tree_B(P, L, Labels)$, $B \in P$ (assuming **choose-next-proposition** will keep making optimal choices, which can be done for example by a complete search of all choices).

Proof: Let us suppose that proposition *A* appears in all labels and so $S_{\varnothing_A} = \varnothing$, while for some other proposition *B*, it might be the case that $S_{\varnothing_B} \neq \varnothing$. Figure 6-8 shows $Tree_A(P, L, Labels)$ (shown at the left side) and $Tree_B(P, L, Labels)$ (shown at the right), $A, B \in P$. Suppose in *Labels* there are π minimal execution scenarios. Let us assume that in S_A , $S_{\neg A}$, S_B , $S_{\neg B}$, $S_{\varnothing_B}$ there are $\varkappa, \lambda, \mu, \nu, \xi$ minimum execution scenarios in each label set respectively (and so $\varkappa+\lambda = \mu+\nu+\xi = \pi$). Then, assuming that **choose-next-proposition** always makes the optimal choice within the subtrees, the subtrees T_1 , T_2 , T_3 , and T_4 each have $\varkappa, \lambda, \mu+\xi, \nu+\xi$ leaves. Thus, if we branch on proposition *A*, $Tree_A(P, L, Labels)$ will have $\varkappa+\lambda = \pi$ leaves, while if we branch on *B*, $Tree_B(P, L, Labels)$ will have $\mu+\nu+2\xi = \varkappa+\lambda+\xi = \pi+\xi$ leaves. Since ξ is always greater or equal to zero, $Tree_A(P, L, Labels)$ has less or equal number of leaves than $Tree_B(P, L, Labels)$. ♦

The theorem indicates that if there is a proposition *A* for which $S_{\varnothing_A} = \varnothing$ at some tree node, then *A* is an optimal choice for branching next. However, it tells us nothing about how to choose the next proposition to branch on if there is no such proposition *A*. We now proceed with a theorem that shows how to make an optimal choice in any case.

Theorem 6-6: Suppose **MakeScenarioTree** (Figure 6-4) is called with parameters *P, L,* and *Labels*, and let Ψ_p as the set of labels in *Unclassified* in which $p \in P$ appears (either positive or negative). A proposition $A \in P$, such that Ψ_A belongs in the minimal cover set of *Labels* by sets Ψ_p is an optimal choice for **choose-next-proposition**, i.e. $Tree_A(P, L, Labels)$ has the smallest number of leaves than any other $Tree_B(P, L, Labels)$, $B \in P$ (assuming **choose-next-proposition** keeps making optimal choices).

Proof: First of all notice that Theorem 6-5 is a special case of Theorem 6-6. Notice that $\Psi_A = S_A \cup S_{\neg A}$ and so if $S_{\varnothing_A} = \varnothing$ then Ψ_A is a minimal cover of *Labels*. Without loss of generality, let us assume that the minimal cover set consists of the two sets Ψ_A and Ψ_B (Theorem 6-5 is a proof when the minimal cover set consists of only one set Ψ_A while the proof below can be extended for any number *n* of sets Ψ_p that make up the minimal cover set).

We now partition the set *Labels* to the following sets:

- $S_{A\varnothing_B}$, the labels that contain *A* but in which *B* does not appear (neither negative or positive)
- S_{AB}, the labels that contain *A* and *B*
- $S_{A\neg B}$, the labels that contain *A* and ¬*B*
- $S_{\neg A\varnothing_B}$, the labels that contain ¬*A* but in which *B* does not appear

- $S_{\neg AB}$, the labels that contain $\neg A$ and B
- $S_{\neg A\neg B}$, the labels that contain $\neg A$ and $\neg B$
- $S_{\varnothing AB}$, the set of labels in which neither A nor B appears

Since, as we assumed Ψ_A and Ψ_B are a minimal cover of *Labels*, then $\Psi_A \cup \Psi_B$ = *Unclassified* and it is the case that $S_{\varnothing AB} = \varnothing$: if $S_{\varnothing AB} \neq \varnothing$ then there would be a label l such that neither A nor B appears and thus $\Psi_A \cup \Psi_B$ could not be a cover of *Labels*. However, for any other set of two propositions C and D, it might be the case that $S_{\varnothing CD} \neq \varnothing$.

By choosing A to branch on next, the left child will get to split the labels $S_{A\varnothing B} \cup S_{AB} \cup S_{A\neg B}$ and the right child the labels $S_{\neg A\varnothing B} \cup S_{\neg AB} \cup S_{\neg A\neg B}$. Let π be the number of minimal execution scenarios in *Unclassified*, and $\varkappa$, λ, μ, ν, ξ, ϱ be the number of minimal execution scenarios in $S_{A\varnothing B}$, S_{AB} , $S_{A\neg B}$, $S_{\neg A\varnothing B}$, $S_{\neg AB}$, $S_{\neg A\neg B}$ respectively. Then, assuming that **choose-next-proposition** will keep making optimal decisions after the first branch, the left subtree will have $\varkappa+\lambda+\mu$ leaves, the right one will have $\nu+\xi+\varrho$ ones, for a total of $\varkappa+\lambda+\mu+\nu+\xi+\varrho = \pi$ leaves.

By considering any other pair of propositions C and D and by choosing C to branch next, the left child will get to split the labels $S_{C\varnothing D} \cup S_{CD} \cup S_{C\neg D} \cup S_{\varnothing CD}$ and the right child the labels $S_{\neg C\varnothing D} \cup S_{\neg CD} \cup S_{\neg C\neg D} \cup S_{\varnothing CD}$. In a similar fashion as above, the total number of leaves in the tree will be $\pi+\sigma$ where σ, the number of execution scenarios in $S_{\varnothing CD}$, is greater than or equal to 0. Note that σ must be added to the total because $S_{\varnothing CD}$ is both included in the left and right subtree. Thus, branching on any proposition C other than A or B, will create a tree with more or equal number of leaves and thus A (or B) is an optimal choice for **choose-next-proposition** at the current node. ♦

The above theorem suggests that one way to optimally **choose-next-proposition** is to calculate the minimal set cover of *Unclassified* by the sets Ψ_p and then pick any proposition for which Ψ_p belongs to the minimal set cover. However, finding the minimal cover set in an NP-complete problem [Cormen, Leiserson et al. 1990] and it will have to be calculated on every tree node. We could of course design an incremental minimal cover set algorithm to reduce the average case time. But, if an approximate algorithm was acceptable, then perhaps a greedy heuristic would be more appropriate. This depends on the optimal trade off point between the cost of including non-minimal execution scenarios in the approximation of the true set of minimal execution scenarios, and the cost or calculating the value of the perfect heuristic.

Based on the discussion and theorems above, we would suggest as the greedy heuristic the one that selects the proposition p that minimizes $S_{\varnothing p}$, in other words the proposition that appears either negative or positive in the most labels. According to Theorem 6-5 this will be an optimal choice when $S_{\varnothing p} = \varnothing$. In addition, as we will see in the next section, when the CSTPs correspond to plans that are produced by a typical conditional planner the greedy heuristic will be optimal. A sketch of a possible alternative approach on generating the set of minimal execution scenarios is in Section 6.11.

6.4 Structural Assumptions for CSTPs

Current conditional planners first branch on some observation A; then all nodes that follow this observation contain in their label either A or $\neg A$ (until the branches merge again if so). Figure 6-10 shows an example of a CSTP that could correspond to such a plan. First node V_1 observes A and depending on the outcome, either nodes $\{V_2, V_4, V_5, V_8\}$ will be executed, or $\{V_3, V_6, V_7, V_9\}$. In a recursive fashion, if A is observed *True* , then B is observed and the execution branches again. If A is observed *False* then C is observed and the execution branches on C. Then, the two branches for B merge and node V_8 is executed if A is *True*, and the branches for C merge and V_9 is executed if A is *False*. Notice how this structure helps the generation of the scenario tree: (i) after we observe a proposition, all subsequent labels contain this proposition, and (ii) the set of observations is ordered. Function **choose-next-proposition** can follow the same order of branches as the plan itself and make optimal decisions.

Another example of a CSTP with structure that could potentially be generated by a conditional planner is at Figure 6-9. This comes from a plan which first branches on A observed by node V_1 , then the two branches merge, and then branch again on B at node V_4

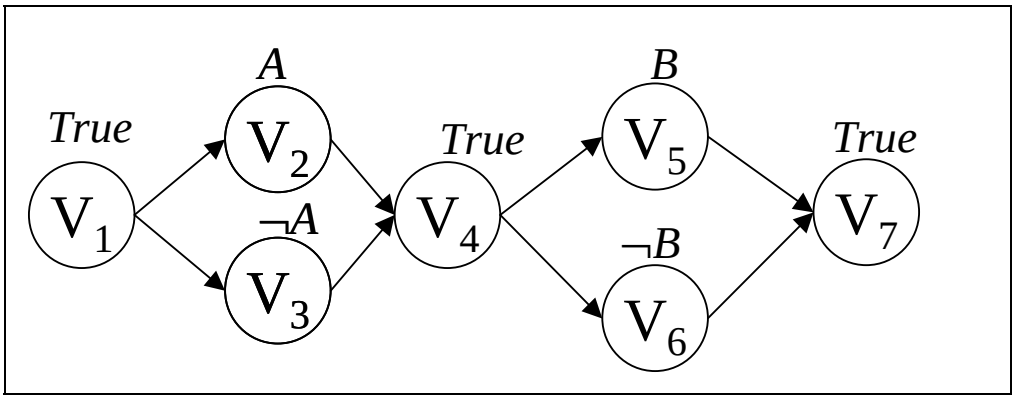


Figure 6-9: A CSTP with benign structure.

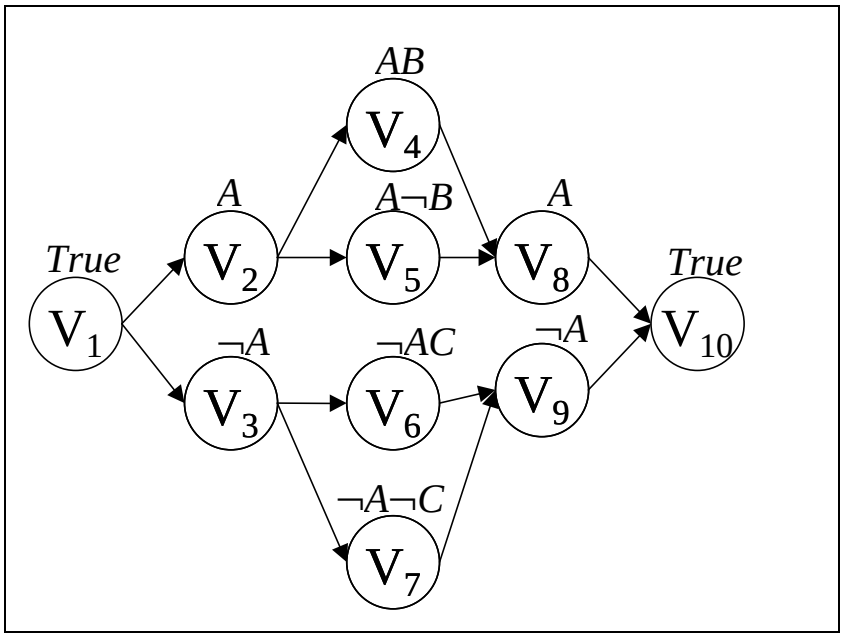


Figure 6-10: A CSTP with typical conditional plan structure.

We now present a couple of examples that look like the above ones, but that in close inspection do not actually exhibit such a structure. The first one is at Figure 6-11, which looks like the one at Figure 6-10, but notice that the label of node V_7 is *True*. That is, even though the plan has already branched on A and C, a label following these observations and before any branch merges, does not contain these propositions in its label. As we will see, this feature complicates the checking of Dynamic Consistency and it is undesirable (it makes no difference though when calculating the scenario tree). The other example, at Figure 6-12, looks again like the one at Figure 6-10. The plan branches on A observed at V_2 and on B observed at V_3. However, these two observations are unordered with each other and they have the same label *True*. In contrast, in Figure 6-10, proposition is observed after A and only when A is observed *True*. The more structured CSTP of Figure 6-10 has four minimal scenarios using three propositions $\{AB, A\neg B, \neg AC, \neg A\neg C\}$, but the CSTP of Figure 6-12 has four minimal scenarios $\{AB, A\neg B, \neg AB, \neg A\neg B\}$ using only two propositions

We now formally define the structural assumptions satisfied by the examples shown at Figure 6-10 and Figure 6-9 that are violated by the CSTPs at Figure 6-11 and Figure 6-12.

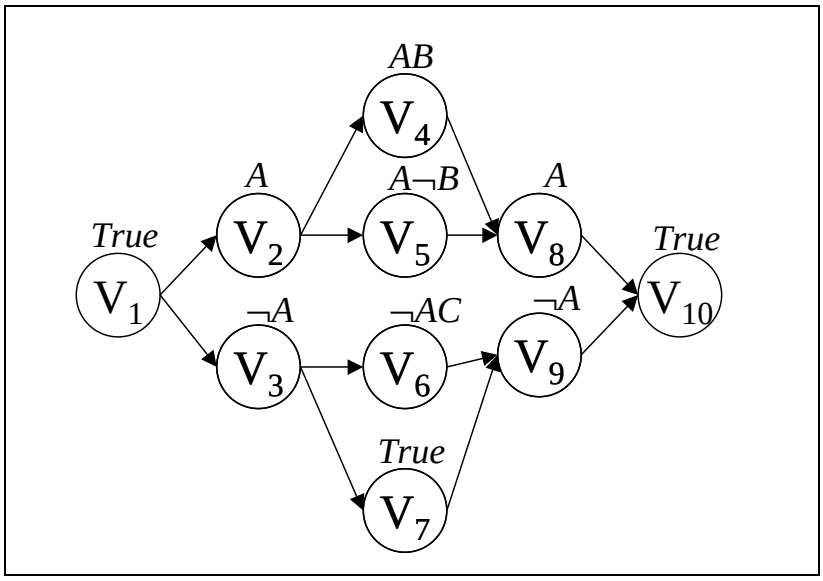


Figure 6-11: An example of a CSTP that does not have typical conditional plan structure

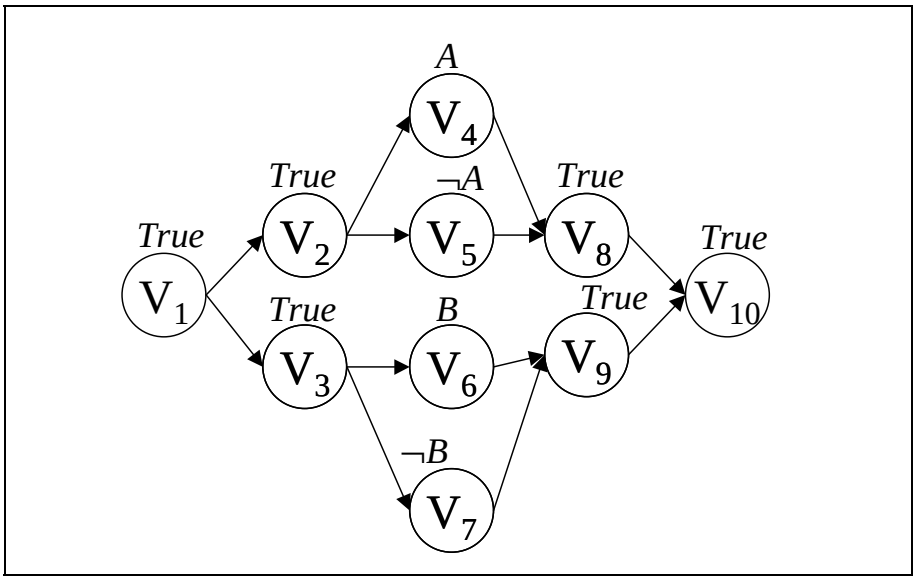


Figure 6-12: An example of a CSTP that does not have benign structure.

Definition 6-18: Let ***D(p)*** for a proposition *p,* be one CSTP node which (i) is constrained to occur after O(*p*) and (just) ***after*** all nodes *n* with label *l* such that *l* contains *p,* and (ii) has label *L(D(p))= L(O(p))*. We will call *D(p)* the ***death of p.*** If there are no such nodes *D(p)* we can insert them artificially into the CSTP. Thus the inequality $O(p) \leq v \leq D(p)$ always holds, where *v* is any node with a label that contains *p*.

Definition 6-19: A CSTP has ***benign structure*** if the following conditions hold for any p and for some possible way of identifying $D(p)$:

1. The set of observation nodes $O(p_i)$ forms a partial order with a single root node.

2. The set of $D(p)$ nodes forms a partial order such that for each pair of propositions p_1 and p_2 either

   a. $O(p_1) \leq O(p_2) \leq D(p_2) \leq D(p_1)$, or

   b. $O(p_1) \leq D(p_1) \leq O(p_2) \leq D(p_2)$, or

   c. Inc $(L(O(p_1)), L(O(p_2)))$

3. If $O(p_1) \leq O(p_2) \leq D(p_2) \leq D(p_1)$, all labels that contain p_2 , also contain either p_1 or $\neg p_1$ (but not both).

If conditions 1, 2, and 3 hold, then for all propositions $p_1 \ldots p_n$ that appear together in any label l, the $O(p_i)$ are totally ordered (same for all $D(p_i)$). Then we can define for a ***label l= $p_1 \ldots p_n$, the observation O(l) and death of a label D(l),*** as $\max(O(p_i))$ and $\min(D(p_i))$ respectively. For the special case label *True* we define $O(True)$ as the node constrained to be first and $D(True)$ as the node constrained to be last (or insert new nodes with the appropriate constraints if there are not such nodes). We can now use this definition for specifying the last condition:

4. For each node $v \in V$, with label $L(v) = l_1$, v is constrained

   a. $O(l_1) \leq v \leq O(l_2)$ or

   b. $D(l_2) \leq v \leq D(l_1)$,

   where l_2 is any label for which $Sub(l_2 , l_1)$.

What does it mean intuitively for a CSTP to have benign structure? The observations are ordered in a partial order (Condition 1), so that two observations are left unordered only if they will never occur together (Condition 2c). Condition 2 specifies that the "life" of a proposition is either contained within the life of some other proposition, or it follows it, or the two propositions are never "alive" in the same scenario. Condition 3 implies that when the life of a proposition p_2 is contained within the life of some other proposition p_1, then p_1 should also appear and specialize the labels that use p_2. Finally, Condition 4 implies that a node v has a label that matches the label of the most specific observation or death node that contains v.

Definition 6-20: A CSTP has ***typical conditional plan structure*** if it has ***benign structure*** and condition 2b never holds.

Definition 6-21: A CSTP has ***typical conditional plan structure without branch merges*** if it has typical conditional plan structure and condition 4b never holds.

As we will see it will be easier to check Dynamic Consistency for CSTPs with the above structure restrictions, than the unrestricted case.

Example 6-4: We now go back to the figures above and check which and why satisfy the definitions above. Figure 6-10 presents a CSTP with typical conditional plan structure. There are ten nodes $V_1 - V_{10}$, and two proposition *A, B,* and *C*. The labels of each node are shown on the top of them while all edges are assumed ordering constraints of the form [0, ∞]. We also define O(*A*) = V_1 , O(*B*) = V_2 , and O(*C*) = V_3 . V_9 is a merge node of the two branches for *C*, and V_{10} the merging node of the branches on *A*. It is easy to identify, $D(A) = V_{10}$, $D(B) = V_8$, $D(C) = V_9$. Also, $O(AB) = O(A\neg B) = O(B) = V_2$, $O(\neg AC) = O(\neg A\neg C) = V_3$ and $D(AB) = D(A\neg B) = D(B) = V_8$, $D(\neg AC) = D(\neg A\neg C) = V_9$. Both conditions 1 and 2 are satisfied (but 2b never holds). For example, for node V_2 with label *L(A)*, $O(A)=V_1 < V_2 < O(AB)$ and $O(A)=V_1 < V_2 < O(A\neg B)$. Had the nodes V_8 , V_9 , and V_{10} been missing the CSTP would additionally have typical conditional plan structure with no merges.

Now consider Figure 6-9. We define O(*A*) = V_1 , O(*B*) = V_4 . The plan first branches on *A*, then the two branches merge, and then the plan branches again to on *B* and merges again. The CSTP has benign structure but not typical conditional plan structure because for propositions *A* and *B* condition 2b holds.

Figure 6-11 shows an example of a plan that does *not* have a benign structure. It is the same as the one at Figure 6-10 but node V_7 now has label *True*, thus the 4th condition of the definition is violated: $L(V_7)=True$, and *Sub(A, True)* and so either (a) $O(True) = V_1 \leq V_7 \leq O(A) = V_1$, or (b) $D(A) = V_{10} \leq V_7 \leq D(A) = V_{10}$ should hold, but none of them holds.

Finally, Figure 6-12 also shows a CSTP that again does not have typical conditional plan structure. In the figure, we define O(*A*) = V_2 and O(*B*) = V_3 . In a first glance the CSTP seems very similar to the one at Figure 6-10. However, the 1st condition of Definition 6-19 is violated: for propositions *A* and *B*, $O(A) = V_2$, $D(A) = V_8$, $O(B) = V_3$, $D(B) = V_9$ but *O(A)* and *O(B)* are unordered with each other.

Theorem 6-7: If a CSTP has typical conditional plan structure, then the greedy heuristic in which **choose-next-variable** always returns the proposition *p* that minimizes $S_{\varnothing_p}$, is an optimal heuristic.

Proof: Let *A* be the proposition for which O(*A*) is the unique parent of the partial order of the observation nodes. Every node *v* has to have a label with (i) no proposition, i.e. *True* , (ii) with one proposition, which has to be *A* since it is the first we observe, or (iii) two propositions, one of which has to be *A* (let us call the other one *B*). In case (iii), because (a) O(*A*) ≤ O(*B*) ≤ *v* holds, and (b) Condition 2b never holds by definition of typical conditional plan structure, then Condition 3 has to hold *n* has to have *A* in its label. Thus, all labels are either subsumed by *True* or contain *A*. Thus, $S_{\varnothing_A} = \varnothing$ and by Theorem 6-5 *A* is an optimal choice.

The arguments above hold recursively for $Labels_{Left} = S_A$ and $Labels_{Right} = S_{\neg A}$ in the **MakeScenarioTree** algorithm if we prove that the Conditions in Definition 6-19 also hold for the labels in $Labels_{Left}$ (similarly for $Labels_{Right}$) and the nodes with these labels. All Conditions 1-4 trivially hold for S_A and the nodes with A in their label (since we are just removing nodes and labels), except Condition 1: the observation nodes in S_A will still be partially ordered, but perhaps with no unique root node. This is impossible as is proved next by contradiction. Suppose that there are two root nodes, O(B) and O(C) that are unordered with each other. By Condition 2c they have to have inconsistent labels. But, since they are the next observations after O(A), A is the only proposition possibly observed and thus their labels can only contain A. Since their labels are inconsistent they have to be A and $\neg A$ which is impossible if they are both in S_A . ♦

6.5 An Algorithm for Determining Weak Consistency

This section presents an algorithm that uses the execution scenario tree built by **MakeScenarioTree** function to determine Weak CSTP Consistency. A brute force algorithm for checking Weak Consistency is to test STP consistency for each *Pr(s)* for every scenario *s*. If all such STPs are consistent the CSTP is Weakly Consistent, otherwise it is not. The execution scenario tree could be used to provide the minimal execution scenarios so that only those are tested for consistency. In addition, we can use the scenario tree to share computation among the STP consistency checks.

Consider the *Pr(ABC)* and *Pr(AB¬C)* of the Example 6-3. The minimum scenario tree is shown at Figure 6-5. Notice that *Pr(ABC)* and *Pr(AB¬C)* share the common "sub-STP", *Pr(AB)*. *Pr(ABC)* is derived from *Pr(AB)* by adding all nodes that are also consistent with *C* and all edges from and to these new nodes to *V(AB)*. Similarly, *Pr(AB¬C)* is derived from *Pr(AB)* by adding all nodes that are also consistent with ¬*C* and all edges from and to these new nodes to *V(AB)*. Thus, if we have determined *Pr(AB)* to be consistent, we can share this computation between the consistency checks for *Pr(ABC)* and *Pr(AB¬C)*: we incrementally determine consistency of *Pr(ABC)* by adding the new nodes and the new edges to *P(AB)* and same for *Pr(AB¬C)*.

Figure 6-13 presents a recursive algorithm that traverses the execution tree and shares as much computation as possible between the STP consistency checks of each *Pr(s)* . The complexity of the algorithm in the worst case is the same as checking STP consistency for each *Pr(s)* from scratch. Recall that each node of the execution tree has a label *L* stored, and left and right child slots. In addition, we will augment the tree nodes with the slot *TimePoints* to store all CSTP nodes that have the same label as the tree node label.

Example 6-5: We will trace the algorithm on the CSTP of Example 6-3 and on the scenario tree of Figure 6-5. The **WeakConsistency** algorithm should first be called with the root of the whole tree, the CSTP, and an empty STP *S*. On the first tree node, labeled with *True*, all the CSTP nodes labeled as *True* are inserted into *S* along with all edges among them. Then, *S*'s consistency is incrementally determined. If it is consistent then **WeakConsistency** is called on the left child with the updated STP *S*. All CSTP nodes labeled *A* are inserted in *S* and all edges among them, and among them and the nodes already in *S*. The traversal of the tree continues in a similar fashion until all nodes have

been visited and proven to correspond to consistent STPs, or at least one of them is inconsistent.

As defined, Weak Consistency requires that all projected STPs *Pr(s)* are consistent. However, if instead of observations we have decision nodes, it suffices to find at least one consistent projection. Alternatively, we might require that only a percentage of projections is consistent, or if we assign probabilities on the observations, that the probability that we encounter a consistent

WeakConsistency(Execution Tree Node T, CSTP C, STP *S*)
1. If *T* = *Nil* Then Return *True* EndIf
2. *T.TimePoints* = {CSTP nodes *n* | label *L(n)* is the same as the label *T.L* of the current tree node)
3. *S'* = The STP *S* with the addition of *T.TimePoints* and all edges from and to the new nodes to the existing ones.
4. Use an Incremental STP checking algorithm on *S'*
5. If *S'* is inconsistent Then
6. Return *False*
7. Else
8. Return **WeakConsistency**(*T.LeftChild* , C, *S'*) ∧ **WeakConsistency**(*T.RightChild* , C, *S'*)
9. EndIf

Figure 6-13: Weak CSTP Consistency checking algorithm.

projection is higher than a given threshold. All these variations are solved by appropriately modifying the main algorithm for determining Weak Consistency.

6.6 Intuitions Behind Dynamic Consistency Checking Algorithms

We now turn to Dynamic Consistency, which is probably the most interesting notion of consistency. It is also, by far, the most complex to analyze theoretically, the most difficult for which to design algorithms, and the most computationally expensive. This section informally discusses the issues that arise trying to design Dynamic Consistency algorithms and develops the necessary concepts and intuitions for the formal discussion that follows. We present a basic example to illustrate all possible cases of CSTP structure that make a difference when calculating Dynamic Consistency.

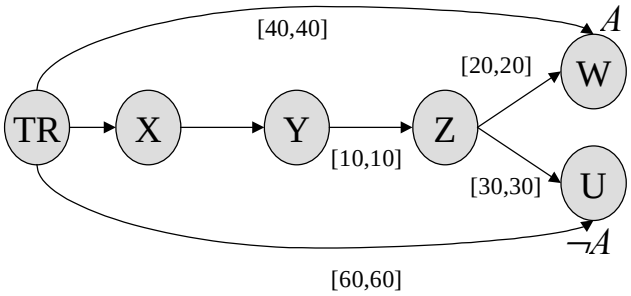
We will use the CSTP of Example 6-1 (Figure 6-1 and Figure 6-14) that we describe here again for ease of presentation. Recall that $V = \{TR, X, Y, Z, U, W\}$ and the labels of the nodes are $L(TR) = L(X) = L(Y) = L(Z) = True$, $L(W) = A$, and $L(U) = \neg A$.

Trivially, there are two (minimal) execution scenarios $s_1 = A$ and $s_2 = \neg A$, and thus two projection STPs $Pr(s_1)$ and $Pr(s_2)$, which are shown in Figure 6-15. In order to distinguish between the same node in different STP projections, we will denote with **N(*v, s*)** the node *v* at *Pr(s)*. For O(*A*), the observation node for *A*, we will use **N(*A, s*)** instead of N(O(*A*), *s*).

In essence, the executor of a CSTN will have to execute one of these projections, but she does not know which one at the beginning of execution. When observation nodes are executed, they provide additional information as to which scenario the execution finally will fall into, but perhaps

the current set of observations does not completely determine the precise STP but a set of STPs with corresponding scenarios consistent with the current observations.

When execution starts, node TR should be assigned a time and in both STNs it should be assigned the same time, since we do not know yet the scenario we will end up with. This means there is an implicit constraint between nodes N(TR, A) and N(TR, ¬A) that constraints them to be executed at the same time. Similarly, if X is providing the observation for A, then N(X, A) and



All edges with no interval are assumed [0, +∞].
All nodes with no label are assumed to have label *True*

Figure 6-14 An example CSTP

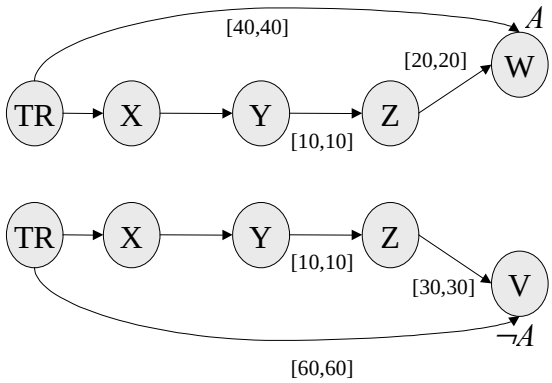


Figure 6-15:The projected STNs for all scenarios

N(X, ¬A) should be assigned the same time, since up to the time when X is executed we do not know which scenario we will be following.

Definition 6-22: A node v is ***synchronized*** among a set of scenarios s_1 , …, s_n when the execution semantics imply that N(v , s_1) = … = N(v , s_n), denoted as **sync(v, s_1 , …, s_n)**.

If *X* is the observation node of the CSTP in Figure 6-14, then *Y* and *Z* do not have to be synchronized because after *X* is executed, we can make a different choice for N(*Y*, *A*) and N(*Y*, ¬*A*) and similarly for *Z*. On the other hand, if *Z* is the observation node of the CSTP in Figure

6-14, then Y and Z do have to be synchronized. In the first case we end up with the STP at Figure 6-16(a) and in the latter one with the STP at Figure 6-16 (b).

Notice that the STP in Figure 6-16 (a) is consistent and the one at Figure 6-16 (b) is not. This suggests that this procedure of translating the original CSTP to an STP – consisting of the projected STPs to all different scenarios with some constraints among them – can be used to determine Dynamic Consistency in the CSTP: if the translated STP is consistent the original CSTN is Dynamically Consistent.

As we will see, deciding when to synchronize is not as easy as suggested so far. Consider for example the constraint between X and Y and assume that it is modified to [-5, 5]. In other words, Y is allowed to occur within 5 time units of the occurrence of X. Also assume that X is the observation node for A. Then, Y should be synchronized if we decide to execute it before X and not synchronized if we decide to execute it after X. In other words, the constraint that should be added between the STPs for the two scenarios is the following:

$$\{(N(Y,A) \leq N(X,A) \vee N(Y,\neg A) \leq N(X,A)) \wedge sync(Y,A,\neg A)\} \vee$$
$$\{N(Y,A) > N(X,A) \wedge N(Y,\neg A) > N(X,\neg A)\}$$

In English, the above formula says that Y can occur before X in all scenarios, in which case Y has to be synchronized between scenarios, **or** Y should occur after X in all scenarios. The above formula can be simplified as:

$$\{(N(Y,A) \leq N(X,A) \wedge sync(Y,A,\neg A)$$
$$\vee \{(N(Y,A) > N(X,A) \wedge N(Y,\neg A) > N(X,\neg A)\}$$

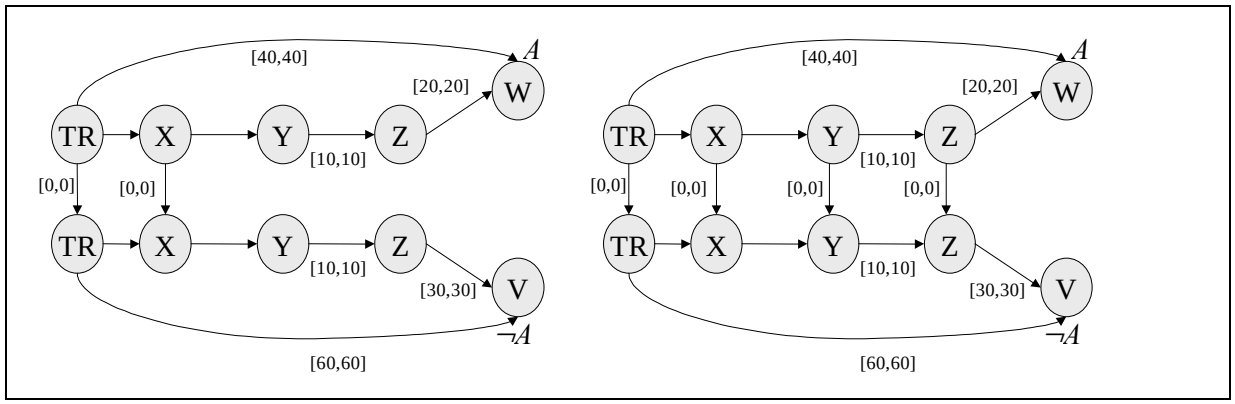


Figure 6-16: The projection STPs with synchronization constraints

We will call such constraints ***DC*** constraints (from Dynamic Consistency). Since DC constraints contain disjunctions, the translation method will result in general to a DTP and not an STP. If there is a solution to the resulting DTP, then the original CSTP is Dynamically Consistent. The problem that arises is how to formalize DC constraints and how to precisely state when they occur.

6.6.1 The problem with unordered observations

A particularly difficult case is when CSTP observation nodes are unordered with respect to each other. For example, if O(*A*) and O(*B*) are unordered with respect to each other, then if O(*A*) occurs first we should synchronize CSTP nodes with label *True* if they occur before O(*A*), otherwise we should synchronize them if they occur before O(*B*). We know proceed with an example. Suppose we have a CSTP with two observations *A* and *B* unordered with respect to each other as shown in Figure 6-17, where O(*A*) = *X* and O(*B*) = *Y* . The exact structure of the network and number of nodes is irrelevant for our purposes and so there might be other nodes in

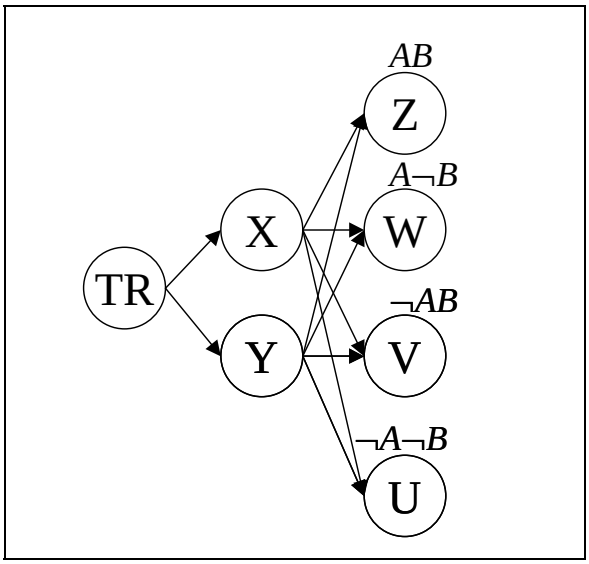


Figure 6-17: A CSTP with unordered observations

CSTP not shown in the figure.

There are four different scenarios for the CSTP,

$$s_1 = AB,\ s_2 = A\neg B,\ s_3 = \neg AB,\ s_4 = \neg A\neg B$$

The projected STPs when O(*A*) < O(*B*) are shown in Figure 6-18(a) and when O(*B*) < O(*A*) in Figure 6-18 (b). All the arrowed edges are ordering constraints [0, ∞] and all non-arrowed edges are synchronization constraints [0, 0]. The scenario of each projected STP is shown at the right side of it. In Figure 6-18 (a) the constraint X < Y is not in the original CSTP, but we display it in the figure because we assumed that O(*A*)=*X* < O(*B*)=*Y* .

What conclusions can we draw from this situation? First of all, after we observe both *A* and *B*, i.e. both *X* and *Y* are executed, we know exactly which scenario we fall into and we can make an independent decision for any N(*v, s*) for any node *v* that comes after *X* and *Y* and any scenario *s*, i.e. we do not need to synchronize *v* in any scenario. More specifically, this means that any node after *Y* in the figure 6-20(a) and any node after *X* in figure 6-20(b) does not have to be synchronized. For any node *v* that appears in all scenarios before either observation, we have no information about which scenario we will be following, and thus *v* has to be synchronized in all scenarios. This is the case for *TR* in both figures and for *X* in figure 6-20(a) and *Y* in figure 6-20(b).

Between these two extremes of not synchronizing in any scenario and synchronizing in all scenarios, there are intermediate situations. In figure 6-20(a), after we observe A by executing X we know whether A is *True* and thus we know we will fall in one of $\{s_1, s_2\}$ or $\{s_3, s_4\}$, but we still do not know with certainty the exact scenario. Thus, in figure 6-20(a) we need to make a common decision about the time we execute Y for both scenarios $\{s_1, s_2\}$ and a common decision for Y for scenarios $\{s_3, s_4\}$. This is why there are the synchronization constraints sync(Y, s_1, s_2) and sync(Y, s_1, s_2) (in the figure shown as solid arrows between the corresponding nodes). In a

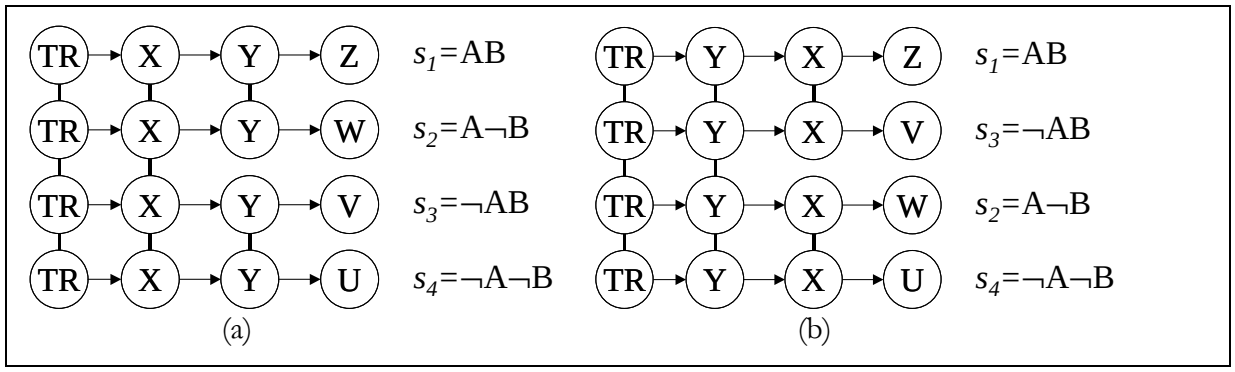


Figure 6-18: The STP projections of the CSTP of Figure 6-17

similar fashion we can see that in when B is observed first as in figure 6-20(b), we have to add the synchronization constrains sync(X, s_1, s_3) and sync(X, s_2, s_4).

Let us suppose now that the CSTP presented in the example has one more node R with label *True* that is underconstrained and can potentially occur in any order relatively to *X, Y, Z, W, V* and *U* (but it has to occur after *TR*). What constraints added to the union of the projected STPs will enforce that a solution to the "super"-STP will guarantee the correct execution of the CSTP under any outcome of the observations? It is either the case that (i) $O(A) = X < O(B)=Y$ or (ii) $O(B)=Y < O(A) = X$ (we are ignoring the equality case for the moment). If (i) then if R is scheduled between *TR* and X it should be synchronized between all scenarios, else if it falls between X and Y is should be synchronized between $\{s_1, s_2\}$ and $\{s_3, s_4\}$, else if it falls after Y it should not be synchronized in any scenario. If (ii) holds instead then similar constrains should be added. We can express these constraints just expressed in English using disjunctions and conjunctions of STP-like temporal constraints.

Let us now summarize the conclusions from this section. The main idea is that we can take the projected STPs for each (minimal) execution scenarios, add temporal constraints among them (disjunctive ones in the worst case, STP-like ones in the best case) and transform the problem to a large DTP problem. The super-DTP will have a solution if and only if the original CSTP is Dynamically Consistent. Moreover, we saw that, for a fixed order of the observation nodes, if a node v is executed in two scenarios before we have enough information to distinguish between the two scenarios, we have to synchronize it. For nodes that can "float" around and that are allowed to be executed either before we can distinguish between the two scenarios or after, we have to add a disjunctive constraint called *DC* that synchronizes the node in the pair of scenarios if we decide to schedule the node before the distinguishing node, or imposes the constraint that the node is executed after this distinguishing point in both scenarios.

Finally, this last subsection in particular, was devoted to the importance of ordering the observation nodes. If observation nodes are unordered with respect to each other, then the constraints that we have to add for one particular order may differ significantly from when we execute these nodes in a different order. Unfortunately, it is our intuition that we have to deal individually with each order of the observation nodes that is allowed by the constraints. For $|P|$ number of propositions this gives a worst case of $|P|!$ possible orderings in the worst case. And for each one such ordering a large number (in the worst case) of disjunctive constraints needs to be added. Thus, in the worst case determining Dynamic Consistency is very hard to compute, even though we do not prove any theoretical result such that the problem is NP-complete.

6.7 Checking Dynamic Consistency

This section provides a general Dynamic Consistency checking algorithm and proves it is correct. The algorithm is exponential in the worst case in the number of propositions and nodes in the CSTP. Here the first algorithm that solves the problem is presented. Our intention here is to focus more in the clarity of the algorithm and ease of proving correctness despite its computational inefficiency. After the general result, we investigate special cases where additional structural assumptions are imposed on the CSTP, resulting in computationally more efficient consistency checking algorithms.

We begin by formalizing the discussion in section 6.6. Let us assume a total order of the observations *order* and consider a node v that appears in ***both*** scenarios s_1 and s_2 (for a node that appears in only one scenario there is no need for the addition of any kind of constraint). In addition, let $O(A)$ be the distinguishing observation, i.e. $O(A)= Dis(s_1 , s_2 , order)$. Trivially, we can infer that $O(A)$ has to be synchronized in the two scenarios: by definition this is the earliest time at which we can distinguish between the two scenarios.. Thus, we should impose the constraint $sync(O(A), s_1 , s_2)$ and since $N(A, s_1) = N(A, s_2)$ we can just define $N(A, s_1) = O(A)$. Then for any other node v we can distinguish among the following cases:

1. $N(v , s_1) < O(A)$ or $N(v, s_2) < O(A)$

   If at least one of $N(v , s_1)$ and $N(v, s_2)$ has to be executed before $O(A)$ then they must be synchronized and must both be executed before $O(A)$.We add the constraint:

   $$sync(v, s_1 , s_2).$$

2. $N(v , s_1) > O(A)$ or $N(v, s_2) > O(A)$

   Similarly, if at least one of $N(v , s_1)$ and $N(v, s_2)$ has to be executed after $O(A)$, they must both be executed after $O(A)$. In this case however, we will know if we are falling into s_1 or s_2 and we can make a different decision for $N(v , s_1)$ and $N(v , s_2)$, i.e. v is left **unsynchronized**. So we should impose the constraints:

   $$N(v , s_1) > O(A) \text{ and } N(v, s_2) > O(A)$$

3. $N(v , s_1) <> O(A)$ and $N(v, s_2) <> O(A)$[22]

[22] We use <> to denote that the left hand side is unordered with the right hand side.

GeneralDynamicConsistency(CSTP *Cstp*)
1. D = **GeneralTranslateCSTP2DTP(*Cstp*)**
2. Solve D by using DTP consistency checking algorithms
3. If D is consistent, return *Dynamic-Consistent*, otherwise return *not-Dynamic-Consistent*

GeneralTranslateCSTP2DTP(CSTP *Cstp*)
4. DTP $D <V, C> = <\cup_i V_i , \cup_i E_i >$, where $Pr(s_i) = <V_i , E_i >$ are the projected STPs
5. For each allowable order $order = O(p_1) \leq \ldots \leq O(p_n)$ of the observations in *Cstp*
6. For each pair of scenarios s_1 and s_2
7. For each node v that appears in both s_1 and s_2
8. $NewC_{v, s1, s2}$= the constraint returned by cases 1, 2, and 3 above
9. $C = C \cup \{order \Rightarrow NewC_{v, s1, s2}\}$
10. EndFor
11. EndFor
12. EndFor
14. Return D

Figure 6-19: The algorithm for determining Dynamic Consistency in the general case.

In this case, we can decide to either execute v before O(A) – in which case we have to synchronize it – or execute it after and leave it unsynchronized. So, we should impose the constraint:

$\{ N(v , s_1) < O(A)$ and $sync(v, s_1 , s_2)\}$ or $\{ N(v , s_1) > O(A)$ and $N(v, s_2) > O(A)\}$[23]

Putting all of our conclusions in use, we present a Dynamic Consistency checking algorithm in Figure 6-19.

The algorithm translates the problem to a DTP and then solves it. If the DTP has a solution, there is a way we can start executing the CSTP so that no matter what the outcome of the observations are, we can continue executing always respecting the constraints, until a complete solution to the CSTP has been found. The translation to a DTP follows directly from the discussion so far: first we add to the DTP all projected STPs and then all necessary additional constraints. Notice in line 9, the constraint $\{order \Rightarrow NewC\}$ encodes the proposition "if *order* holds then *NewC* holds".

Computationally, it is possible to increase performance by first computing all distance arrays of the projection STPs using the **WeakConsistency** of Figure 6-13. If any of the projected STPs is inconsistent, the CSTP is obviously not Dynamically Consistent. Moreover, by having all these distance arrays available we can check whether $O(A) < O(B)$ for two observation nodes in every scenario, and thus not iterate for any *order* in line 5 inconsistent with our findings. In addition, for any two scenarios s_1 and s_2 it is easy to check in which of cases 1, 2, or 3 we fall when the distance arrays for $Pr(s_1)$ and $Pr(s_2)$ are available. Finally, notice that *NewC* (line 8) will be the same for all

[23] Actually, this third case subsumes the other two: we could return the constraint of the third case in all three cases.

orders of observations that give the same $Dis(s_1, s_2, order)$, providing us with more opportunities for simplifying the added constraints.

Theorem 6-8: Algorithm **GeneralDynamicConsistency** is correct.

Proof: It is easy to see that an execution strategy *St* corresponds to an exact schedule *T* of *DTP D* (line 3) by defining that for every scenario $s \in \boldsymbol{Sc}$, then $St(s) = T_s$, where T_s is the projection of *T* to *Pr(s)* (i.e. $T_s(v) = T(N(v, s))$). Thus, an execution strategy defines an exact schedule *T* for *D* and vice versa.

Case (a): Let us suppose that **GeneralDynamicConsistency** returns *Dynamically-Consistent*; we will prove the input *Cstp* is Dynamically Consistent.

Let *T* be a schedule that is an exact solution to the DTP *D* and *St* the corresponding execution strategy. *St* is a viable execution strategy because *T* is a solution of every *Pr(s)* since it is a sub-STP of *D* (line 4). *T*, as an exact schedule, induces a total order *order* to the observations of $Pr(s_1)$ and $Pr(s_2)$ for any s_1 and s_2 and so, from the definition of the distinguishing observation, it must be the case that $Dis(s_1, s_2, T_{s1}, T_{s2})$ is the time of the observation node $Dis(s_1, s_2, order)$ in either T_{s1} or T_{s2}. Now, in all cases 1, 2, and 3 above, the constraints added dictate that if a node *v* is ordered before $Dis(s_1, s_2, order)$ in any of the two scenarios, then it has to be synchronized between the two; thus $T_{s1}(v) = T_{s2}(v) \Rightarrow St(s_1)(v) = St(s_2)(v)$.

Case (b): Let us suppose that **GeneralDynamicConsistency** returns *non-Dynamically-Consistent*; we will prove by contradiction that the input *Cstp* is not Dynamically Consistent.

Suppose that *Cstp* is Dynamically Consistent and thus there is a viable *St* that satisfies the definition and a corresponding exact schedule *T* for *D*. We will prove that *T* satisfies all the constraints of the translated DTP and so the latter is consistent so the algorithm would not have returned *non-Dynamically-Consistent*. It is easy to see that since *St* is viable, $St(s)=T_s$ is a solution to *Pr(s)* and so *T* respects all the constraints added at line 4 of the algorithm. What is left to prove is that *T* also respects all constraints added at line 9 of the algorithm.

For any pair of scenarios s_1 and s_2 , *T* defines a specific observation node $Dis(s_1, s_2, T_{s1}, T_{s2}) = T(\mathrm{O}(\mathcal{A}))$ so that $\mathrm{O}(\mathcal{A})=Dis(s_1, s_2, order_T)$, for some order of the observation nodes. *St* satisfies the conditions of the definition of Dynamic Consistency, and thus $\forall\ v \in V$, if $T_{s1}(v) \le Dis(s_1, s_2, T_{s1}, T_{s2})$ or $T_{s2}(v) \le Dis(s_1, s_2, T_{s1}, T_{s2})$, then $T_{s1}(v) = T_{s2}(v)$, also written as $N(v, s_1) = N(v, s_2) \Rightarrow \mathrm{sync}(v, s_1, s_2)$. The above statement can be written as:

$$\{N(v, s_1) < \mathrm{O(A)} \text{ and } \mathrm{sync}(v, s_1, s_2)\} \text{ or } \{ N(v, s_1) > \mathrm{O(A)} \text{ and } N(v, s_2) > \mathrm{O(A)} \} \quad (1)$$

for the particular $order_T$, $\mathrm{O}(\mathcal{A})$, s_1 , and s_2 .

The algorithm, at each iteration *k* of the three loops adds the constraint $\{order_k \Rightarrow NewC\}$. If $order_T \neq order_k$ this latter constraint is trivially satisfied. If $order_T = order_k$ then *NewC* holds because equation (1) above holds for $order_T$ and it is equivalent to *NewC* (see case 3). ♦

6.7.1 Dynamic Consistency checking for CSTPs with structural assumptions

Why is checking Dynamic Consistency so hard? As we have shown, when observations are unordered with each other, we have to try all possible different orderings. Another source of intractability occurs when nodes are unordered with respect to observations. In that case, we need to add disjunctive constraints instead of conjunctions of STP-like constraints. Finally, a third source of complexity is the need to represent each node v with a large number of nodes N(v, s), one for each scenario v appears in.

There are, however, CSTPs in which all these three intractability factors disappear. What is the complexity of checking Dynamic Consistency in those structures? Surprisingly, it is the same as treating the original CSTP as an STP! Thus, for this special class of CSTPs Strong Consistency and Dynamic Consistency degenerate to the same concept. What kind of structural assumptions do we have to make to obtain this result? In this section we will prove that for CSTPs with typical conditional plan structure with no merges Dynamic Consistency is equivalent to STP consistency.

Lemma 6-1: Let v be a node with label $l = p_1 \ldots p_n$ in a CSTP with benign structure. Then (for some appropriate indexing of p_i) $O(p_1) \leq \ldots \leq O(p_n)$.

Proof: For each $O(p_i)$, $O(p_j)$ it cannot be the case that Inc($L(O(p_i))$, $L(O(p_j))$) or the observations would not be both executed in the same scenario as v (which makes the CSTP not well defined). Thus, Condition 2c of Definition 6-19 never holds and so every pair of observation nodes must be ordered (by Conditions 2a and 2b of Definition 6-19). ♦

Lemma 6-2: In a Dynamically Consistent CSTP $<V, E, L, OV, O, P>$ with benign structure, typical conditional plan structure, or typical conditional plan structure with no branch merges, for every pair St_1 , St_2 of successful execution strategies, and for every pair of scenarios s_1 , $s_2 \in \boldsymbol{Sc}$, $Dis(s_1, s_2, order_{S1}) = Dis(s_1, s_2, order_{S2})$, where $order_{S1}$, $order_{S2}$ are the order of observation nodes the St_1 and St_2 induce on s_1 and s_2 respectively.

Proof: If the lemma were false, that would mean that there is an observation node $O(A)=Dis(s_1, s_2, order_{S1})$, and another node $O(B) = Dis(s_1, s_2, order_{S2})$, so that $O(A) < O(B)$ in St_1 and $O(B) < O(A)$ in St_2 . That means that Condition 2c of Definition 6-19 must be the case, or the observation nodes would have to be ordered with each other by Conditions 2a and 2b.

Thus the two observation nodes have inconsistent labels. Let us suppose that the two labels have only one proposition in them, proposition C, and so they must be C and $\neg C$ to differ. But, this means that there should be an observation $O(C)$ before both $O(A)$ and $O(B)$ that separates the two scenarios and thus $O(C)$ is the distinguishing observation, and not $O(A)$ or $O(B)$. We can generalize the argument when the labels of $O(A)$ and $O(B)$ have more than one proposition. Say $L(O(A)) = p_1..p_n z_1..z_m$ and $L(O(B))= p_1..p_n q_1..q_m$, thus z_1 and q_1 the first two observations that differ in their labels. Since these two observations must have been performed before both $O(A)$ and $O(B)$ one of them (the one scheduled the earliest) would have been the distinguishing observation, and it would not be neither $O(A)$ or $O(B)$ as we assumed. ♦

Lemma 6-3: Let CSTP $<V, E, L, OV, O, P>$ be one with typical conditional plan structure with no branch merges. Then, for every node $v \in V$, and for every pair of scenarios s_1 , s_2 in which v is executed in (i.e., $Sub(s_1, L(v))$ and $Sub(s_2, L(v))$) , it is the case that sync(v, s_1 , s_2) has to hold, i.e. every node has to be synchronized in all scenarios that it appears in.

Proof: Suppose that the lemma is false. That means that there are two different scenarios s_1 , s_2 : $Sub(s_1, L(v))$ and $Sub(s_2, L(v))$ in which v does not have to be synchronized. For all p_i that appear in $L(v)$ a total order has to hold for all of O(p_i) (Lemma 6-1), i.e. $L(v) = p_1..p_n$ (the p_i can be either positive or negative). Since $Sub(s_1, L(v))$ and $Sub(s_2, L(v))$, then s_1 and s_2 are labels of the form $s_1 = p_1..p_n q_1..q_m$ and $s_1 = p_1..p_n z_1..z_k$, i.e. they must also contain $p_1..p_n$. This in turn implies that the outcome of the observation nodes for all of these propositions has to be the same in both s_1 and s_2 . Thus, $O(L(v))$ (see Definition 6-19 for the definition of O(l) where l is a label) cannot be the distinguishing observation no matter what the order of the rest of observations.

Since $O(L(v))$ is not the distinguishing observation, and s_1 , s_2 are different scenarios, there has to be some other distinguishing observation node O(A) for some order of the rest of the observations (O(A) has to be synchronized in s_1 and s_2). Let $l_2 = L(v) \wedge A$ (we could also have used $\neg A$). Remember that condition 4b does not hold when we do not allow branch merges, thus 4a must hold and so, since $Sub(l_2 , L(v))$ it must be the case that $O(L(v)) \leq v \leq O(l_2)$. In other words, for any distinguishing observation O(A), v is constrained to occur before it, and so it has to be synchronized. ♦

Now we are ready for the punch line!

Theorem 6-9: Checking Dynamic Consistency in a CSTP with typical conditional plan structure with no branch merges is equivalent to checking STP consistency on the CSTP, ignoring the label and observation information.

Proof: By Lemma 6-2 we know that for any pair of total orderings of the observation nodes, consistent with the partial order ***Par*** in Condition 1 of Definition 6-19, $Dis(s_1 , s_2 , order_{S1}) = Dis(s_1, s_2, order_{S2})$, and so the constraint produced by the algorithm **GeneralTranslateCSTP2DTP** (line 9) is the same constraint $NewC_{v, s1, s2}$ for all orders consistent with ***Par*** or it is an inconsistent order. Thus, GeneralTranslateCSTP2DTP has to produce constraints $\{order \Rightarrow NewC_{v, s1, s2}\}$ only for *order* consistent with ***Par*** (all other ones are trivially satisfied since they are inconsistent and thus the right hand side of the implication is false) and the $NewC_{v, s1, s2}$ will be independent of the particular order because for any order $Dis(s_1 , s_2 , order_{S1}) = Dis(s_1 , s_2 , order_{S2})$. Thus, we can simplify this constraint to just $\{NewC_{v, s1, s2}\}$. To see this, assume we have the constraints $\{a \Rightarrow c\} \wedge \{b \Rightarrow c\}$ and we know either a or b has to hold. We can simplify this set of constraint to just c .

Now, by Lemma 6-3 it is the case that $NewC_{v, s1, s2}$ has to be a synchronization constraint (Case 1 in the analysis before the presentation of the algorithm). Moreover, since it is the case that for any pair s_1, s_2, where v appears sync(v, s_1 , s_2) holds, then transitively, sync(v, s_1 , .., s_n) holds, where s_1 , .., s_n are all the scenarios where v appears. In other words, we can

collapse all nodes N(v, s) since they always occur at the same time. What we end up with then, is an STP, which is the same as the CSTP with all context information, removed. ♦

Unfortunately, when a CSTP does not exhibit typical conditional plan structure with no branches, Lemma 6-3 does not hold, and it might be the case that some node v will have to be represented with more than one N(v, s) in the translated temporal network returned by **GeneralTranslateCSTP2DTP**. For example, node V_7 in Figure 6-11 will have to be represented by N(V_7 , s) for all four scenarios. Similarly, V_5 in Figure 6-12 has to be represented by both N(V_5 , AB) and $N(V_5$, $\neg AB)$. However, even in all of these cases the resulting network will still be an STP.

When are we guaranteed to end up with an STP instead of a DTP during the translation process of a CSTP? It is easy to see from cases 1-3 that the nodes have to be ordered with respect to the observation nodes. So, as long as for any node v , and any observation O(A) that appear together in any scenario, it is the case that O(A) < v or vice versa, the resulting network will still be an STP. This condition also implies that any pair of observation nodes, that appears together in some scenario, is ordered, and thus the observation nodes have a partial ordering.

Theorem 6-10: GeneralTranslateCSTP2DTP will return an STP if for any node v , and any observation O(A) that appear together in any scenario, O(A) and v are ordered with respect to each other. ♦

Finally, even when **GeneralTranslateCSTP2DTP** returns a DTP instead of an STP, the DTP may still be practical if the outer loop of the algorithm runs for only one iteration, i.e. the constraint $NewC_{v, s1, s2}$ (line 9) does not depend on the *order* of the outer loop. For this to be true, all observation nodes that appear in the same scenario have to be ordered with respect to each other so we can use the same arguments as in the proof of Theorem 6-9 that line 9 may only add the constraint $NewC_{v, s1, s2}$, instead of { $order_k \Rightarrow NewC_{v, s1, s2}$ } and not iterate for *order* .

Theorem 6-11: If for every pair of observation nodes O(B), O(A) they are ordered with respect to each other with the same order in all scenarios they both appear, then **GeneralTranslateCSTP2DTP** need not iterate over *order* and may just add the constraint $NewC_{v, s1, s2}$, instead of { $order_k \Rightarrow NewC_{v, s1, s2}$ }. ♦

6.8 Disjunctive Conditional Temporal Networks (CDTP)

It is easy to see, that in the above discussion regarding the CSTPs, we never assumed that the constraints E are STP-like constraints. Thus, the same arguments and algorithms hold even when the constraints have the general form of DTP-like constraints.

Definition 6-23: A ***Conditional Disjunctive Temporal Network (CDTP)*** is defined similarly to a CSTP where the constraints E can be DTP-like constraints.

6.9 A Related Approach

In this section we compare our approach with the only other approach that we know of that combines quantitative temporal constraints and alternative contexts, i.e. the TCN networks of Barber [Barber 2000]. TCNs are based on TCSPs allowing a limited form of disjunctions among

the time-points. Barber describes an extension to scheme that annotates each TCSP-like disjunction with a context label W_i. Having different contexts permits a partition of the timeline to a set of independent chains. The contexts are arranged in a hierarchy such that constraints can be associated with different contexts. An example of this representation scheme is shown at Figure

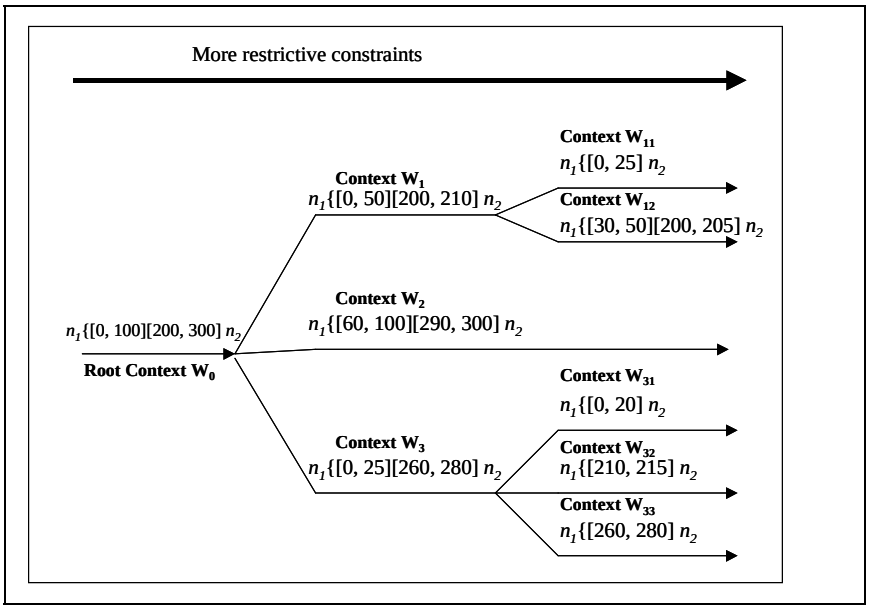


Figure 6-20: A Temporal Constraint Network with context information [Barber 2000]

6-20 (reproduced from [Barber 2000] p.73) where the displayed TCN has two time-points (variables) n_1 and n_2 and contexts W_0, W_1, W_2, W_3, W_{11}, W_{12}, W_{31}, W_{32}, and W_{33}. In the figure we use his notation: n {[a, b], [c, d]} m expresses the constraint ($a \leq m - n \leq b \vee c \leq m - n \leq d$). Successor contexts of a given context represent different alternatives, which are mutually exclusive, e.g. W_{11} and W_{12} are mutually exclusive. The constraints in a context W_i also hold in all successor contexts W_{ij}, but not in predecessor contexts nor among contexts of different hierarchies. The definition of consistency given for such a TCN is:

> **Definition 6-24: "**A context-dependent TCN is minimal (and consistent) if the constraints in each context are consistent (with respect to constraints in this context, in all its predecessor contexts, and all it's successor contexts) and minimal (with respect to constraints in this context and in all its predecessor contexts)." [Barber 2000]

Using the above definition and the TCN of Figure 6-20, we can construct a hierarchy of TCSPs shown at Figure 6-21 (again reproduced from [Barber 2000]). Every node in the tree of the figure corresponds to a TCSP with the constraints that belong to the set of contexts shown in the node. The TCN is consistent if and only if each such TCSP is consistent[24]. A comparison of Barber's TCNs and CSTPs and CDTPs is discussed in the last section of this chapter.

It is unclear how the context hierarchy is created in Barber's approach. The planning system, or in general the system in need of temporal reasoning, is responsible for creating this hierarchy by

[24] As we saw, a similar kind of trees also appeared when we calculate Weak CSTP Consistency.

using domain knowledge on what differentiates possible future outcomes. In conditional planning for example, the splitting (branching) to different more specialized successor contexts succeeds an observation. Thus, in Figure 6-20 we can imagine there is an observation associated with every branch. Now consider the situation where we have another new plan, similar to the one in Figure 6-20, with its own observations and contexts that we would like to merge with the current one. How does the new context hierarchy look like? Unless we know how each context is related to the observations and how the observations are related with each other, we cannot automatically create the context hierarchy. However, in CSTPs, one can use the algorithms for creating the execution tree and automatically create the equivalent of Barber's context hierarchy. This is because in CSTPs the contexts are directly related to observations.

Another disadvantage of the TCN as defined by Barber, is that it does not provide a way of guaranteeing the correct dispatch and execution of plans represented as TCN. If an executive is executing the TCN of Figure 6-20 then it should assign times to the time-points so that no matter what future context occurs, there is a way to proceed with the execution in such a way that it respects the temporal constraints. Barber does not define the equivalent of Dynamic Consistency, but only provides a consistency definition analogous to Weak Consistency.

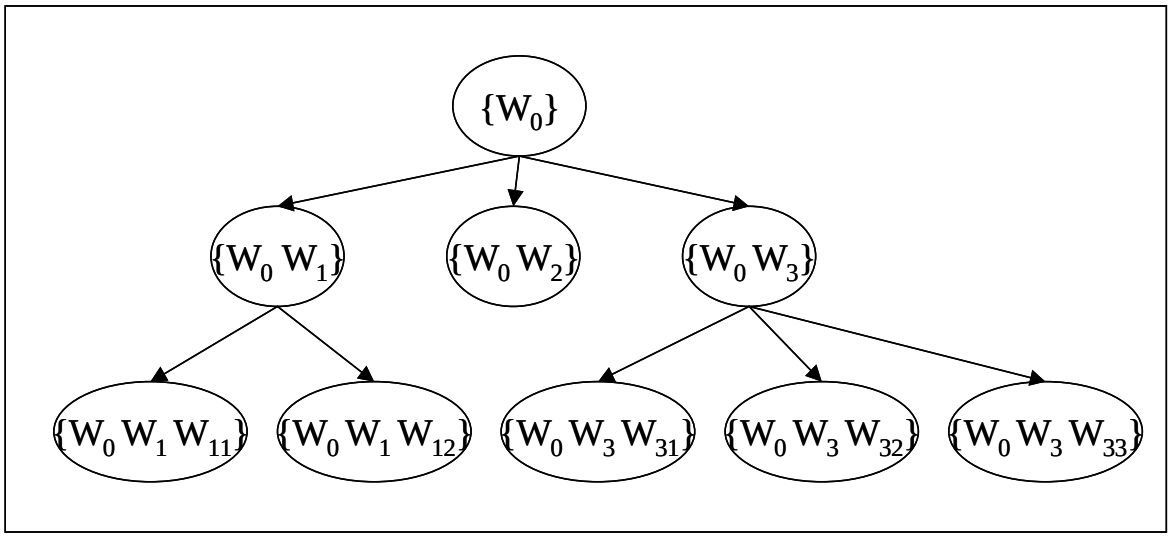


Figure 6-21: The hierarchy of induced TCSPs of Figure 6-20 (reproduced from [Barber 2000])

6.10 Encoding Conditional Constraints

The CSTP and CDTP definitions only allow the representation of conditional actions, i.e. actions that are only executed when certain conditions are true; the definition does not attach conditions (labels) to constraints and thus it does not directly allow the representation of conditional constraints, i.e. constraints that should only be respected when their conditions (label) are true. We now describe how to circumvent this problem and represent such constraints in CSTP and CDTP.

First we consider the CSTP case where the constraints are non-disjunctive and we will use a representation trick to encode conditional constraints in CSTPs. For each conditional constraint C of the form C: $x - y \leq b$ that we want to represent and that should be respected only when the label $L(C)$ is true, we insert two dummy CSTP variables w and z with labels $L(w) = L(z) = L(C)$. We also insert the constraints $w - z \leq b$, $0 \leq x - w \leq 0$, and $0 \leq y - z \leq 0$ so that x and w have to co-occur, and similarly for the pair y and z. Notice now that when the label $L(C)$ becomes true, the dummy nodes w and z have to be executed and thus the constraint $w - z \leq b$ has to hold between them. Because w and z co-occur with x and y respectively, this means that when $L(C)$ is true (and only then), the constraint $x - y \leq b$ holds between them. In other words, what we achieved is to represent constraint C using only conditional actions and not conditional constraints. The latter feature would require changing the CSTP definition and the reasoning algorithms. Schematically, the idea in the above discussion is displayed at Figure 6-22.

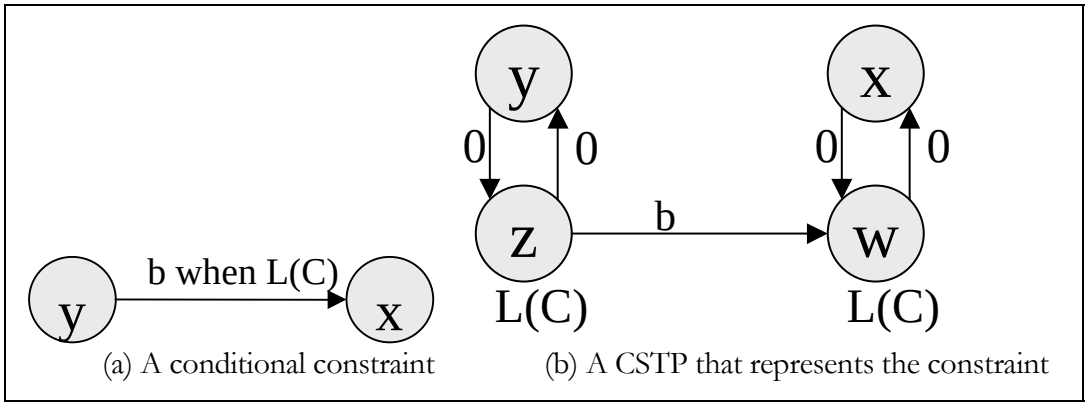


Figure 6-22: Representing conditional constraints in CSTPs.

Representing disjunctive, n-ary conditional constraints in CDTPs is similar, but more than two dummy variables are required. For example, we could represent the constraint $x - y \leq b_1 \vee s - t \leq b_2$ with condition (label) l, by creating the dummy nodes w, z, u, v all with label l, and insert the constraints $w - z \leq b_1 \vee u - v \leq b_2$, $0 \leq x - w \leq 0$, $0 \leq y - z \leq 0$, $0 \leq s - u \leq 0$, and $0 \leq t - v \leq 0$.

6.11 An Alternative View on the Problem of Calculating the Set of Minimal Execution Scenarios

We have given considerable attention to the problem of calculating the set of minimal execution scenarios. As we saw, the algorithm for finding strictly minimal scenarios is computationally intractable in general (although not for CSTPs with specific structural assumptions). We thus presented a heuristic approach based on a greedy selection of branching variables. In this section, we present another approach, using (Ordered) Binary Decision Diagrams. We have not yet formalized this approach completely and the purpose of this section is to present our intuitions behind this approach, hoping to motivate other researches to continue this line of work too.

We attempt to formulate the problem of calculating the set of minimal execution scenarios to the problem of computing a minimal representation of a multi-valued binary function. This calculation can efficiently be performed using Ordered Binary Decision Diagrams (BDD)[Somenzi

1999]. BDDs are compact representations of Boolean formulas and relatively recently they have found their way from systems verification and model checking [Burch, Clarke et al. 1994; Yang, Bryant et al. 1998] to planning [Jensen and Veloso 2000]. BDDs have been given a lot of attention in systems' verification and a number of efficient systems have been implemented and are publicly available.

For a CSTP <*V, L, E, ON, O, P*> let us define

$$f : \{True, False\}^m \rightarrow \{True, False\}^n : f(p_1, \ldots, p_m) = (l_1, \ldots, l_n)$$

where l_i are the labels associated with the nodes.

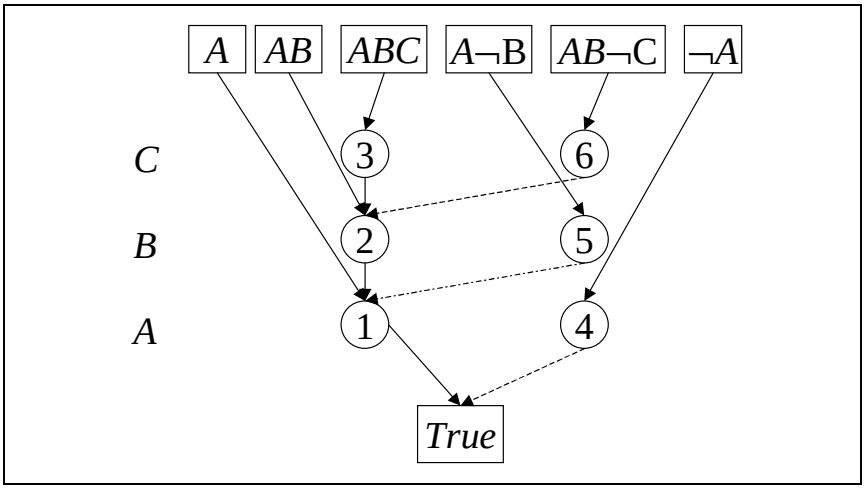


Figure 6-23: The minimum BDD for function *f*.

Function *f* maps each complete assignment of the propositions p_1 , …, p_m to a vector of *True* and *False* with a value for each node label and thus partitioning the nodes to those whose label becomes *True* and those whose label becomes *False*. If a node *v* is in a projection STP for a scenario *s* , *Pr(s)*, then its corresponding *L(v)* in the output vector of *f(s)*, where *s* is a complete assignment to the propositions, is *True*. Since for two equivalent scenarios s_1 , s_2 it is the case that $Pr(s_1) = Pr(s_2)$, then $f(s_1) = f(s_2)$. Without a proof, we intuitively expect that *finding the minimal Boolean representation for f is equivalent to finding the set of minimal execution scenarios.*

Example 6-6: Let us assume that the set of labels are {*A, ¬A, AB, A¬B, ABC, AB¬C*}. Then *f(A, B, C) = (A, ¬A, AB, A¬B, ABC, AB¬C)*. The BDD of Figure 6-23 is the minimum BDD for *f*.

In a BDD the top nodes in the squares correspond to single-output functions, in our case the set of labels l_1 , … , l_n . Each level of the BDD contains testing nodes for a specific proposition, e.g. the top level contains the nodes for proposition *C* and the bottom one for *A*. Each node performs a test on that proposition and if the proposition is *True* it branches to the next node with a solid arrow. If it is *False* it branches with a dashed arrow. Any path from the top level function to the square marked with *True* specifies the conditions for the function to become *True*. For example, the third function from the left, *ABC*, is *True* when path ABC → 3 → 2 → 1 → *True* is followed, meaning that *C, B,* and *A* all have to be true for the function to be true. Function *AB*¬*C* is *True* when the path *AB*¬*C* → 6 → 2 → 1 → *True* is followed, meaning that it becomes *True* when *C* is *False* (because the arrow emanating from 6 is dashed) , *B* is *True,* and *A* is *True*.

The size of the BDD is greatly influenced by the order of the tests of the propositions. One can see the relationship between Figure 6-5 and Figure 6-23: the order of the testing of the variables is *A, B, C* in Figure 6-5and *C, B, A* in Figure 6-23. Moreover the have the same

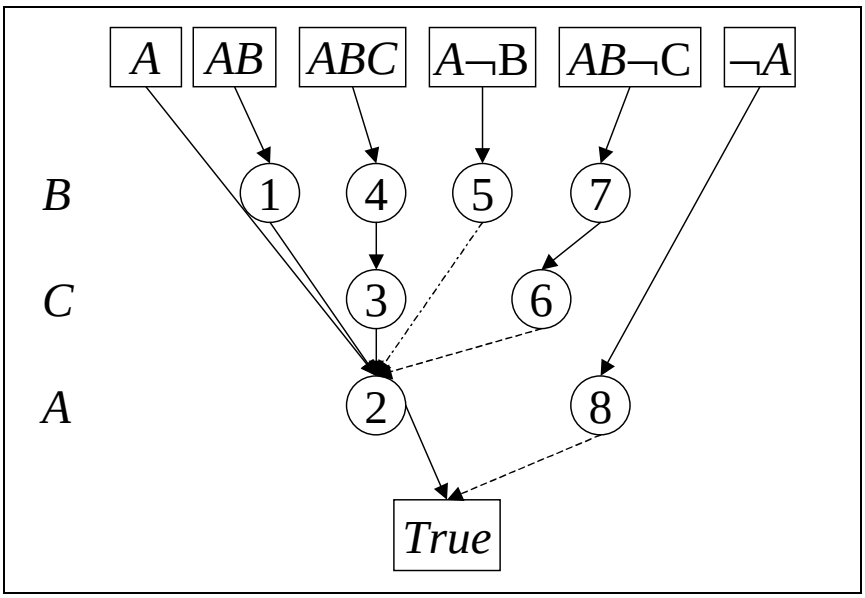


Figure 6-24: The BDD for *f* with the order *B, C, A*.

number of nodes (if we include the *True* node in Figure 6-23). Figure 6-24 shows the BDD for the order of the propositions *B, C, A* that corresponds to the order *A, C, B* of Figure 6-7. Again, Figure 6-24 and Figure 6-7 have the same number of tree nodes and BDD nodes respectively for the same order of propositions.

The problem now is once one has the minimum BDD representation of *f*, how to extract the set of minimum execution scenarios to which we have no answer at the moment, but we hope, given the similarities of the two approaches, to soon solve. Notice that in any case, if our conjecture is true, one could at least use the BDD packages, and all the work done on exact and approximate proposition ordering heuristics, to find the optimal (or a good approximation) testing order of the propositions that minimize the BDD representation of *f* , and then use that for the function **choose-next-variable**.

We also tried to map the problem of calculating the minimal execution scenarios to a decision tree problem, given the similarities of the execution tree building approach and the decision tree approaches. However, unlike the supervised machine learning problems solved by decision trees, the labels are not classified to classes, which is essential to all decision tree algorithms when deciding which attribute (proposition) to branch next and when to stop branching.

6.12 Discussion and Contributions

In this chapter we presented two new classes of problems, namely CSTP and CDTP that permit reasoning with alternative contexts, observations and conditional courses of action. Moreover, the two formalisms can also encode quantitative temporal constraints, the latter one including disjunctions.

The contributions of this chapter are summarized as follows:

- We presented formal definitions of CSTP and CDTP.
- We identified and defined three notions of consistency, namely Strong Consistency, Weak Consistency, and Dynamic Consistency.
- We proved the result that Weak Consistency in CSTP is co-NP-complete.
- We defined a minimal execution scenario as a minimal set of observations that define which nodes should be executed and which should not, and provided algorithms for calculating the set of minimal execution scenarios without having to generate all the possible scenarios.
- We proved that checking Strong Consistency is equivalent to checking STP consistency.
- We provided an algorithm for checking Weak Consistency that, unlike the obvious brute force algorithm, performs incremental computation.
- We provided a general algorithm for determining Dynamic Consistency.
- We provided special cases of CSTPs where additional structural assumptions make the generation of the set of minimal execution scenarios computationally easy.
- For one special case of CSTPs, that we called CSTPs with typical conditional plan structure with no merges, we proved that Dynamic Consistency is equivalent to STP consistency of the CSTP.
- We determined other factors and structural assumptions that facilitate the determination of Dynamic Consistency.

Finally, we would like to note the use of DTPs as a theoretical and practical tool in checking Dynamic Consistency. Without a good understanding of DTPs it might have been impossible to design algorithms for Dynamic Consistency. A different approach that could be used is in [Vidal 2000] where temporal problems are translated to time automata. We believe that this approach has theoretical merits but it is not practical since it creates automata with an exponential number of states in the size of the CSTP and observations.

7. PLANNING WITH TEMPORAL CONSTRAINTS

This chapter applies the constraint-based temporal reasoning paradigm to planning in order to solve several planning problems in a conceptually clear way and potentially increase efficiency. The main result in this chapter are algorithms for the problems of plan merging, plan cost optimization, and plan cost evaluation in context for richly expressive plans with quantitative temporal constraints and conditional branches by casting them as DTPs or CDTPs problems.

7.1 A Motivational Scenario and a Presentation of Basic Concepts

7.1.1 Plans and plan schemata

Let us consider a clinic in a hospital with doctors, equipment, resources, and, of course, patients. There are a number of activities in the hospital taking place every day, such as patient admissions, medical tests being performed on patients, doctors examining patients, patient discharges, papers moving from office to office, appointments being made, etc. These activities can be grouped together according to the purpose they serve. For example the actions of: (i) arranging an appointment with a patient John Smith, (ii) reserving the MRI on the second floor for 1 hour starting 3pm, Monday for Smith, (iii) performing a specific test with the MRI, (iv) sending the results to Dr. Rob Brown, (v) mailing bill #1234 to Joe Smith's insurance company, and finally (vi) getting paid, all serve the purpose of performing the MRI test on patient John Smith. These actions are related to each other with causal relations (e.g. performing the test provides the results to send to the doctor), and all support the achievement of the same goal or set of goals. In addition, there might be certain constraints among the actions, for example, that performing the test must take place after reserving the equipment for it, or that the bill must be mailed to the insurance company within 20 days after the test. Conceptually, such a set of related actions, events, causal relationships, and constraints forms a ***plan*** or a ***plan instance***.

At any point in time, the system – the whole clinic in this example – has committed to a set of plans that is executing. Some of these may look very similar to each other, such as two plans for performing a test on patient *X* and a test on patient *Y*. Both plans contain the same type of actions even though some of them concern different patients or different doctors. We can think of the two plans being derived from the same skeletal description of performing a test on some patient, in which the patient, the specific equipment used, or the doctor that evaluates the results is not specified. We will call these descriptions ***plan schemata***. The steps for the plan schema *MRITestSchema* for the example plan above are (i) arrange an appointment with patient X, (ii)

reserve MRI Y for 1 hour at time Z, (iii) perform the test at time Z, (iv) send the results to doctor Q, (v) mail bill R to X's insurance company (vi) get paid. The constraints are that X is a patient, Y is an MRI machine, Z is a time point, Q is a doctor, R is a bill, all actions should occur in the order presented, and the MRI reservation should be for an hour. Plans are therefore instantiations of plan schemata with some or all of the missing details filled in with specific information; or, equivalently, plan schemata are abstractions of plans where specific information is replaced with ***variables***[25]. In a clinic, there are a finite and usually relatively small number of plan schemata, but a potentially infinite number of potential plan instances.

7.1.2 Executing plans

In any system that involves actions we can distinguish between a decision-making subsystem, the ***planner***, who decides the kind of actions must be performed, and an action taking component, the ***executor***. Currently, in most clinics like the one in the example above, the planner and the executor are formed by people who take the decisions and subsequently act. The work in this dissertation aims at providing tools for increasing the automation of the planner component, and therefore, it could be used in systems in which either the executor is a human or a set of humans (like in the clinic domain), or in completely autonomous systems in which both the planner and the executor are fully automated [Muscettola, Nayak et al. 1998].

The plans can be thought of achieving a set of ***goals*** (e.g. having a specific test performed). Obviously, the people who specify the medical procedures in the clinic of the example have taken great care in ensuring their correctness. We require that *every possible execution* that respects the constraints, of *every possible instantiation* of a plan schema, must indeed achieve the goal for which it is intended. If this is not true then perhaps more constraints or a more detailed model of the world should be added to the schema.

Therefore, to execute a given plan, the executor has to find a solution to the set of constraints C in the plan. A solution is an assignment of exact times to the actions to be taken, and an assignment of the variables to constants (e.g. doctor X bound to doctor Rob Jones). For the plan that performs the test in the example above, a solution would assign times to the actions so that they occur in the right order, and bind the variables to specific doctors, patients, MRI equipment etc. So for example, performing the test might be assigned the time of 4pm and sending the results to the doctor the time of 4:30pm. Notice though, that if the test takes more than half an hour such an assignment of times to actions will cause an incorrect execution of the plan since the results will not be ready in time. This is why a solution to the set of constraints C of a plan is typically constructed dynamically during execution, this way taking into account uncontrollable events (in this case the ending of the test action). Only when the test is over, are the results sent to the doctor and the action assigned an execution time.

When the constraints are just ordering constraints as in this example, finding a solution to a constraint set is easy and does not require any complex or time consuming reasoning. However, with other types of constraints (such as quantitative temporal constraints), obtaining a solution

[25] We would like to note here that the variables in the plan are distinct from the variables in the STP (or DTP or CDTP) that will be used to represent it. Some of the former (i.e. the temporal variables; in the example Z) will be mapped to the latter, but non-temporal variables, e.g. X or Y will not.

might require sophisticated reasoning (as we demonstrated in Chapter 5), especially when done dynamically and in real time [Tsamardinos 1998]. Typically, for any plan, there are many or even infinite choices for execution; the set of constraints may leave many details unspecified or underspecified, and hence the problem is under-constrained. Therefore, the question that arises is why not constrain the plan schemata in such a way that every derived plan has only one solution, and present the executor with it, so that she will not have to reason herself. The answer to the question is that the resulting plans are very brittle. Any small deviation from the specified solution will break some constraint in the plan and correct execution would not be guaranteed anymore. As above, if the test is scheduled for 4pm and the next action for 4:30pm, the test has to finish by 4:30pm. If every aspect of the domain is modeled, there is no uncertainty, and the planning agent is omniscient, we could then predict everything that could happen during execution and present the executor with one solution instead of a set of constraints that describes a set of solutions. Unfortunately, these assumptions hardly ever hold in the real world and therefore a plan should always allow as much flexibility as possible for executing it.

7.1.3 Identifying and resolving conflicts

To summarize the above discussion, we begin with a set of plan schemata that guarantee that every execution of every plan instantiation will achieve the goals of the plan as long as the execution respects the constraints in the plan. Nevertheless, this is true only when plans are considered in isolation. When two or more of them are jointly considered for execution, potential negative interactions (***conflicts***) may occur. Negative interactions depend on the semantics of each step **and** on whether the current plan constraints allow them to interact. Some actions for example should not overlap, some others need to be chronologically separated by some time period, and others should not occur in a specific order. So for example, two steps that may use the same type of equipment conflict if they are allowed to overlap; the administration of drug A and drug B to the same patient creates a conflict if the two drugs are incompatible with each other and they need to be chronologically separated for some period of time but they are not constrained to. Finally, a step that dictates that a robot should move to the coffee room for the purpose of getting coffee for a user, conflicts with a step that moves the robot to another room that could be executed between going to the coffee room and before fetching the coffee.

Suppose the executor of the system is told to execute two plans for performing two different tests on two different patients. The plans will both be instantiations of the plan schema *MRITestSchema* for performing a test. To correctly execute the join of the plans, it is not enough for the actors to respect the union of their constraints: it is possible that they both attempt to reserve the same MRI for the same time because nothing constrains that possibility. The two actions of reserving the MRI conflict with each other. *Intuitively, two actions conflict with respect to a set of constraints C, when C allows executions in which the two actions negatively interact.*

As requests for tests, patient admissions, etc. arrive in the clinic, new plans will need to be executed in the context of the current plans or commitments. To guarantee correct execution it is required that conflicts are identified and resolved. Conflicts are resolved by imposing additional constraints to the set of current plans, called ***conflict resolution constraints*** ; these restrict the set of possible executions further to filter out the ones in which actions negatively interact. We will

see that these constraints can be represented by a DTP or extensions of a DTP. For example to resolve the conflict between the two MRI reservation actions (let us name them X and Y) we could impose the constraint that "X's reservation time ends before Y's reservation time begins ***or*** Y's reservation time ends before X's reservation time begins ***or*** X reserves a different MRI than Y does" (These constraints are directly related to promotion, demotion, and separation in classical planning [Weld 1994]). Additionally, conflicts could be resolved by adding new steps in the plans but we will not consider this option in this chapter and leave it for future work.

It is worth examining in more detail the role and nature of conflict resolution constraints with the aid of the following example. Suppose that actions A, B, and C, which belong to different plans, all reserve and use the same equipment X for exactly one hour between 1pm and 5pm. As the situation is stated, nothing prevents A, B, and C from overlapping during execution, and so there is a conflict for each pair of actions: <A, B>, <B, C>, <A, C>. A way to resolve them could be by imposing three constraints, one for resolving each conflict: "A should finish before B starts or B should finish before A starts", "B should finish before C starts or C should finish before B starts" and "A should finish before C starts or C should finish before A starts". Using an intuitive shorthand notation, this can be also written as "A→B or B→A", "B→C or C→B", and "A→C or C→A". In other words, there is one constraint for each pair or actions that conflict to disallow them from overlapping, and thus remove the conflict. Adding the above constraint to the existing ones means that A, B, and C should be executed between 1pm and 5pm, take exactly one hour, and not overlap with each other. Notice that, as is typical with conflict resolution constraints, there is more than one option for resolving a particular conflict, meaning the constraints are disjunctive. The above constraints can be represented as a DTP and executed as such with the algorithm we presented in Chapter 5; alternatively, a solution component STP to the DTP can be found and the plan could be executed as an STP instead.

7.2 The Plan Merging Problem

7.2.1 Plan representation

We now define the plan representation we will be using. It is based on classical partial-order planning representations [Weld 1994] but with significant extensions such as disjunctive quantitative temporal constraints. It is straightforward to extend our plan-merging algorithms to use other representations such as those based on Constraint-Interval Based (CBI) planners [Smith, Frank et al. 2000].

For the moment we consider only propositional plans, i.e. plans in which the predicates do not take any parameters[26]. We will discuss how to extend our algorithms to include parameterized predicates in Section 7.4.

We represent plans as tuples of the form $P = \langle S, T, L \rangle$, with the components defined as follows:

[26] The algorithms in this section can trivially be generalized to plans with predicates that take variables, but whose variables are all bound to constants at merging time: for each predicate *P(X)* , where *X* is an object, define the proposition *P-X*; the resulting plans then become propositional and given as input to the algorithms presented in this chapter will produce the correct output.

- S is a list of the ***plan steps***. As usual in the planning literature, each step is associated with a particular operator, which has ***preconditions*** and ***effects***. Since we will be considering temporal plans, actions have temporal extent and are not assumed instantaneous as in classical planning. We will assume that preconditions have to hold before the action begins ***and*** during the action, and that the effects hold as soon as the action starts ***and*** continue to hold after the action until changed by some other action. This is a conservative approach: in general a precondition may not have to hold until the action finishes execution and respectively an effect may not be immediately produced when the action starts execution. Because of the temporal extent of the actions, we can enrich our representation with ***in-conditions*** and ***resources***. In-conditions are predicates (fluents) that become true during the action (step), and resources are predicates that correspond to single-capacity re-usable resources that are in use while the action is being executed. E.g. *Printer-A* might be a resource of an action, indicating that the printer with name *A* is in use during the action's execution. As described in Section 2.1.1 and Chapter 5, we associate two time-points with each step, one representing its start and another representing its end, we will use the notation $Start(s_i)$ and $End(s_i)$ for a step s_i to denote them (in the figures we will use the shorthand $S(s_i)$ and $E(s_i)$). Similarly to the conservative treatment of preconditions and post-conditions, in-conditions and resources are used throughout the duration of a step. It is not hard to modify this scheme for less conservative approaches by introducing additional time-points that represent the exact the moment an effect is produced or a precondition or a resource is not required any more.
- T is a set of ***temporal constraints***. Elements of T are DTP-like constraints on the time-points $Start(s_i)$ and $End(s_i)$ for all $s_i \in S$, and a special *time reference point* TR associated with some arbitrary time. We will assume that for every step $s_i \in S$, T contains the constraint $Start(s_i) - End(s_i) \leq 0$ constraining its duration to be greater or equal to zero.
- L is a set of ***causal links***, defined in the typical classical planning way as triplets $<s_p, e, s_c>$, where e is both an effect of s_p and a precondition of s_c. Step s_p is the *producer* of the causal link and s_c is the *consumer* of the causal link. We assume that T contains the constraint $End(s_p) - Start(s_c) \leq 0$ so that the producer finishes execution before the consumer starts.

7.2.2 Identifying conflicts

We now present the different types of conflicts that might exist between the steps of a plan $P = <S, T, L>$. We will denote with $s_1 \rightarrow s_2$ the constraint that s_2 is after s_1, i.e. $End(s_1) - Start(s_2) \leq 0$.

Definition 7-1: A ***Causal Conflict*** between steps s_p and s_t in plan $P = <S, T, L>$ exists if the following conditions hold:

1. $<s_p, e, s_c> \in L$ for some step s_c in S.
2. There is *another* step s_t with effect $\neg e$.
3. Step s_t might begin before step s_c ends, i.e. $Start(s_t) - End(s_c) \leq 0$ is consistent with T.
4. Step s_t might end after step s_p ends, i.e. $Start(s_p) - End(s_t) \leq 0$ is consistent with T.

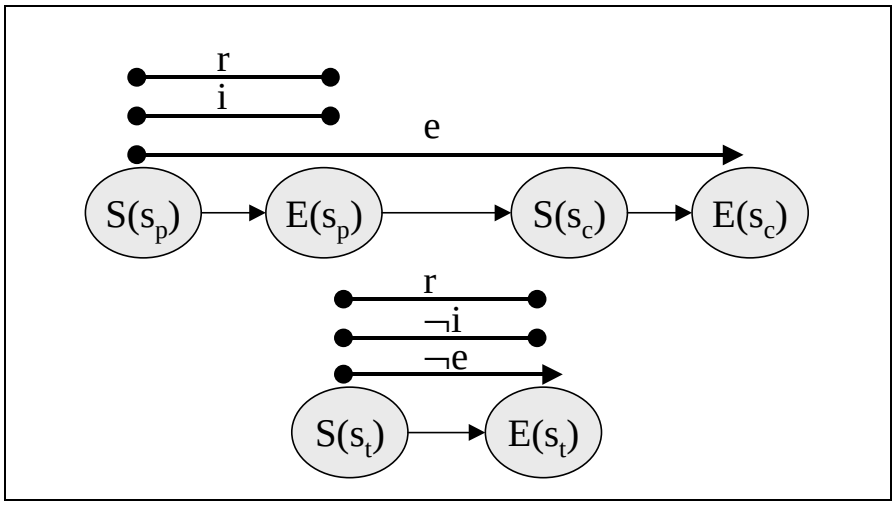


Figure 7-1: Possible conflicts between pairs of steps.

5. There is no other step s_k with effect e, that necessarily comes between s_t and s_c , i.e. so that $s_t \rightarrow s_k \rightarrow s_c$ necessarily holds.

A causal-conflict is shown pictorially in Figure 7-1 where six time-points are shown corresponding to the start and end times points of steps s_p , s_t , and s_c . Step s_p has the effect e , the in-condition i , and uses resource r . Step s_c has precondition e and step s_c has a precondition e and step s_t the effect $\neg e$, the in-condition $\neg i$, and uses resource r. We also assume the causal-link $<s_p, e , s_c>$ is part of the plan.

The causal link implies that predicate e has to hold just before and during the execution of consumer s_c. Thus, it either (i) has to hold uninterrupted from the moment s_p starts execution and e is produced, until s_c is completed and e is not required by s_c any more, or (ii) another step s_k has to re-establish e for consumption by s_c. . If neither of (i) or (ii) is true, then there is a causal conflict. The given definition for causal conflict generalizes the notion of threats in classical plans, in which conditions 3 and 4 would be replaced by a simpler requirement that s_t may possibly come between s_p and s_c . A similar definition of conflicts appears in the *IxTeT* system [Ghallab and Laruelle 1994] and [Yang 1992].

Definition 7-2: An ***In-Condition Conflict*** between two different steps s_p and s_t in plan $P = <S, T, L>$ exists if the following conditions hold:

1. Step s_p has an in-condition i.
2. Step s_t has an in-condition $\neg i$.
3. Steps s_p and s_t might overlap, i.e. $Start(s_t) - End(s_p) \leq 0$ and $Start(s_p) - End(s_t) \leq 0$ are both consistent with T.

An in-condition specifies a predicate that the step makes true and requires to be true during its execution. Thus, if two steps have inconsistent in-conditions they should not overlap. In Figure 7-1 this is shown pictorially for the in-conditions i and $\neg i$ and steps s_p and s_t respectively.

Definition 7-3: A ***Recourse Conflict*** between two different steps s_p and s_t in plan $P = <S, T, L>$ exists if the following conditions hold:

1. Steps s_p and s_t both have the same resource r.
2. Steps s_p and s_t might overlap, i.e. $Start(s_t) - End(s_p) \leq 0$ and $Start(s_p) - End(s_c) \leq 0$ are both consistent with T.

Our plans are inherently parallel: all steps that are not constrained to occur at different times may be executed simultaneously. The use of resources makes it straightforward to prohibit co-occurrence of steps that would have harmful interactions with one another. The use of resources has turned out to be important for many of the steps we have modeled in intelligent workflow/calendar management system we have been building [Tsamardinos, Pollack et al. 1999]. For example, in modeling the step of "having a meeting", it is important to record the fact that the agent is busy throughout the duration of the meeting, and therefore is prohibited from scheduling another meeting at the same time. This can readily be done by attaching a resource *busy* – the attention of the person attending the meeting – to the "have a meeting" operator.

7.2.3 Defining and solving the plan merging problem

Our goal is to take two plans expressed in the representation language described above, and find temporal constraints that ensure their consistency. The definition of this problem can trivially be generalized for N number of plans.

Definition 7-4: Let $P_1 = <S_1 , T_1 , L_1 >$ and $P_2 = <S_2 , T_2 , L_2 >$. The ***plan-merging*** problem is to find a set of temporal constraints T' such that the plan $P' = <S_1 \cup S_2 , T_1 \cup T_2 \cup T', L_1 \cup L_2 >$ can be executed in a conflict-free way, i.e. any solution to $T_1 \cup T_2 \cup T'$ achieves all the goals.

As described in Section 7.1 P_1 will represent the conjunction of the agent's existing plans, and P_2 will represent its plan for a new goal. The plan-merging problem has been addressed for the case of classical plans by Yang[27] in [Yang 1997] and we will discuss the differences and similarities with our approach in the related research section below.

Notice that the above definition of plan-merging restricts us to resolve the conflicts that arise by only imposing additional temporal constraints. Thus, the possibility of changing the set of steps by adding or removing steps or sets of steps is excluded as a way of conflict resolution.

Let us now examine the different ways available to resolve the conflicts in the plan P. We first consider causal conflicts. To resolve a causal conflict between steps s_p and s_t we can invalidate any of the conditions in Definition 7-1. We can invalidate condition 3 by adding the constraint $\neg(Start(s_t) - End(s_c) \leq 0)$, which is equivalent to $End(s_c) - Start(s_t) < 0$, and if we are not concerned with the transition point where $End(s_c) = Start(s_t)$, the previous constraint can be written as $End(s_c) - Start(s_t) \leq 0$ (so as to be in DTP format). Similarly, we can invalidate condition 4 by adding the constraint $End(s_t) - Start(s_p) \leq 0$. The former constraint corresponds to promotion of the clobberer in classical planning and the latter to demotion.

It is also possible to invalidate condition 2 if s_c and s_t are the same type of step, provided that we allow ***step-merging*** and merge the steps s_t and s_c so that semantically they refer to the same step.

[27] What we describe here as "plan merging" is referred to as "global conflict resolution" by Yang: it is the process of finding constraints that guarantee that two plans do not negatively interact with one another. Yang reserved the term "plan merging" to refer to the process of combining two or more *steps* of the same type. We prefer to this latter process as "step merging".

The idea in step-merging is that two steps of the same type are combined into one. For example, two steps of type *Go-From-Home-To-Store* could be combined into one such step. This is not the case for all types of steps, e.g. two steps of type *Take-Dosage-of-Drug-A* should not be merged since taking two dosages of a drug is inherently different than taking only one dosage of the same drug. We will assume that the domain encoding contains a function $Mergable(s_1, s_2)$ that returns true if the two steps are allowed to be merged. Notice that we cannot invalidate condition 2 by step merging s_t and s_p since they have effects $\neg e$ and e respectively and thus they cannot be of the same type.

Finally, we can invalidate condition 5 by enforcing that another step s_k with effect e comes between the clobbering step s_t and the consumer s_c. Step s_k is called the "white knight" in classical planning.

Any of the above ways for invalidating the conditions of Definition 7-1 is adequate to resolve the conflict. Thus, we can impose the disjunctive conflict resolution constraint:

Conflict-Resolution Constraint for causal-conflict between s_p, s_t and causal link : $<s_p, e, s_c>$

$$s_c \rightarrow s_t \ \vee$$
$$s_t \rightarrow s_p \vee$$
$$(Mergable(s_t, s_i) \wedge Merge(s_t, s_c)) \vee$$
$$\vee_{s \in S_e} \ s_t \rightarrow s \wedge s \rightarrow s_c$$

where S_e is the set of steps with effect e and $Merge(s_1, s_2)$ is a shorthand for the constraint $Start(s_t) = Start(s_c) \wedge End(s_t) = End(s_c)$. Following the terminology of classical planning, the first disjunct corresponds to promotion, the second to demotion, the third to step-merging, and the disjuncts generated by the equation $\vee_{s \in S_e} \ s_t \rightarrow s \rightarrow s_c$ to the introduction of white-knights. The third disjunct in the above conflict-resolution constraint can be simplified at constraint-generation time at either *False* if $Mergable(s_1, s_2)$ is *False*, or $Merge(s_t, s_i)$ if it is *True*. Note that after this simplification the constraint is in DTP format. Temporally, we denote merging two steps by synchronizing their starts and ends*, but the fact that we decided to merge the steps should also be recorded and communicated to the executor*.

The treatment of in-condition conflicts and resource conflicts is similar:

Conflict-Resolution Constraint for in-condition-conflict between s_p, s_r and in-condition *i*:

$$s_p \rightarrow s_t \ \vee \ s_t \rightarrow s_p$$

Notice that we cannot merge the two steps to resolve the conflict in this case, because by definition the two steps have conflicting in-conditions and thus it is impossible for them to be of the same type.

Conflict-Resolution Constraint for resource-conflict between s_p, s_t and resource *r*:

$$s_p \rightarrow s_t \ \vee \ s_t \rightarrow s_p \vee (Mergable(s_p, s_i) \wedge Merge(s_p, s_t))$$

Definition 7-5: We define as ***conflict-resolution constraints*** of two plans P_1 and P_2 ***(CRC_{12})*** the union of all the DTP conflict-resolution constraints for all causal, in-condition, and resource conflicts.

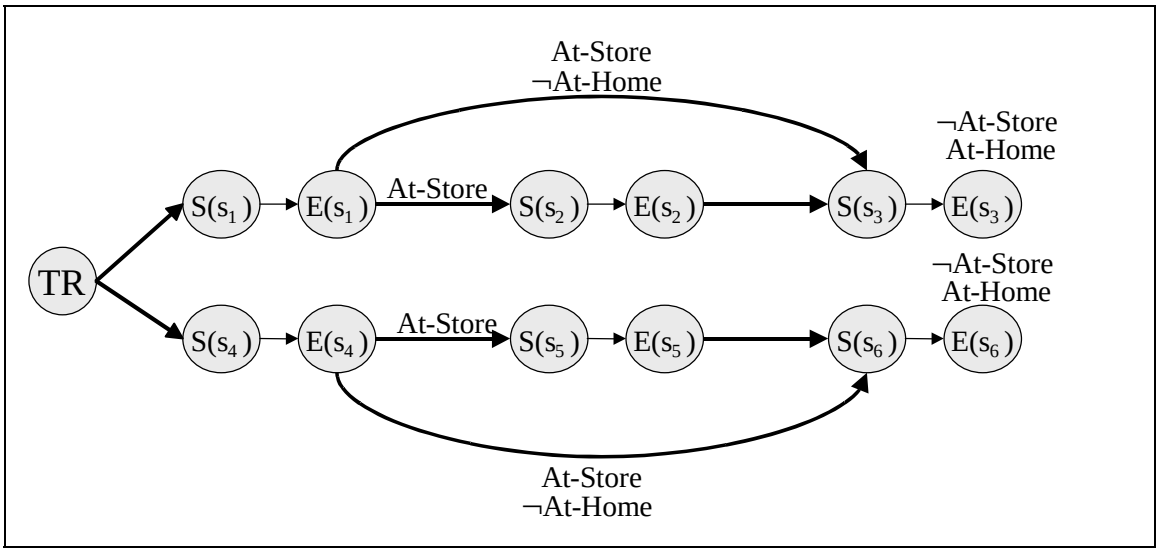


Figure 7-2: Example of causal-conflict resolutions

We now illustrate the different ways of resolving causal conflicts with the simple example below:

Example 7-1: Figure 7-2 shows two plans P_1 (at the top part) and P_2 (at the bottom part) being merged. P_1 involves going to the store, buying milk, and returning to home, and P_2 involves going to the store, buying fruit, and returning to home. $P_1 = <S_1 , T_1 , L_1 >$ and $P_2 = <S_2 , T_2 , L_2 >$, where:

- $S_1 = \{s_1 = Go\text{-}From\text{-}Home\text{-}To\text{-}Store , s_2 = Buy\text{-}Milk , s_3 = Go\text{-}From\text{-}Store\text{-}To\text{-}Home\}$
- T_1 are the constraints $s_1 \rightarrow s_2$ and $s_2 \rightarrow s_3$.
- $L_1 = \{<s_1 , At\text{-}Store , s_2 > , <s_1 , At\text{-}Store, s_3 >, <s_1 , \neg At\text{-}Home, s_3 >\}$ (generating the ordering constraints shown with the bold edges in the figure).
- $S_2 = \{s_4 = Go\text{-}From\text{-}Home\text{-}To\text{-}Store , s_5 = Buy\text{-}Fruit , s_6 = Go\text{-}From\text{-}Store\text{-}To\text{-}Home\}$
- T_2 are the constraints $s_4 \rightarrow s_5$ and $s_5 \rightarrow s_6$.
- $L_2 = \{<s_4 , At\text{-}Store , s_5 > , <s_4 , At\text{-}Store, s_6 >, <s_4 , \neg At\text{-}Home, s_6 >\}$

Also, the steps *Go-From-Home-To-Store* (s_1 and s_4) have effects *At-Store* and $\neg$*At-Home* and no preconditions[28], the steps *Go-From-Store-To-Home* (s_3 and s_6) have preconditions *At-Store* and $\neg$*At-Home* and effects *At-Home* and $\neg$*At-Store*. The steps *Buy-Milk* (s_2) and *Buy-Fruit* (s_5) have preconditions *At-Store*. Finally, the steps *Go-From-Home-To-Store* and *Go-From-Store-To-Home* are allowed to be merged together.

There are six causal-conflicts:

1. s_1 conflicts with s_6 because of causal-link $<s_1 , At\text{-}Store , s_2 >$
2. s_1 conflicts with s_6 because of causal-link $<s_1 , At\text{-}Store, s_3 >$
3. s_1 conflicts with s_6 because of causal-link $<s_1 , \neg At\text{-}Home, s_3 >$, and symmetrically

[28] We do not define preconditions for these steps because of complications that would arise with the initial state. We discuss this issue again later on this section. Also, the example does not show for simplicity the causal links to dummy goal actions with preconditions *Have(Milk)* and *Have(Fruit)*.

4. s_4 conflicts with s_3 because of causal-link $<s_4 , At\text{-}Store , s_5>$
5. s_4 conflicts with s_3 because of causal-link $<s_4 , At\text{-}Store, s_6>$
6. s_4 conflicts with s_3 because of causal-link $<s_4 , \neg At\text{-}Home, s_6>$

The conflicts generate respectively the CRC_{12} constraints below:

1. $s_6 \rightarrow s_1 \vee s_2 \rightarrow s_6 \vee (s_6 \rightarrow s_4 \rightarrow s_2)$
2. $s_6 \rightarrow s_1 \vee s_3 \rightarrow s_6 \vee Merge(s_3 , s_6) \vee (s_6 \rightarrow s_4 \rightarrow s_3)$
3. $s_6 \rightarrow s_1 \vee s_3 \rightarrow s_6 \vee Merge(s_3 , s_6) \vee (s_6 \rightarrow s_4 \rightarrow s_3)$
4. $s_3 \rightarrow s_4 \vee s_5 \rightarrow s_3 \vee (s_3 \rightarrow s_1 \rightarrow s_5)$
5. $s_3 \rightarrow s_4 \vee s_6 \rightarrow s_3 \vee Merge(s_3 , s_6) \vee (s_3 \rightarrow s_1 \rightarrow s_6)$
6. $s_3 \rightarrow s_4 \vee s_6 \rightarrow s_3 \vee Merge(s_3 , s_6) \vee (s_3 \rightarrow s_1 \rightarrow s_6)$

Since $T_1 \cup T_2 \cup CRC_{12}$ has at least one solution (in fact there are three), CRC_{12} is a solution to the plan merging problem of P_1 and P_2 .

At this point it is necessary to have a closer look to the plan-merging problem. First of all, notice the introduction of steps s_4 and s_1 as white-knights generating the last disjunct in constraints 1-3 and 4-6 respectively. As we have mentioned there are three STP solutions to the DTP defined by the constraints $T_1 \cup T_2 \cup CRC_{12}$. The first solution, described intuitively, is to first buy the fruit, return home, then go again to the store and buy milk. This corresponds to selecting the disjuncts $\{ s_6 \rightarrow s_1 , s_6 \rightarrow s_1 , s_6 \rightarrow s_1 , s_5 \rightarrow s_3 , s_6 \rightarrow s_3 , s_6 \rightarrow s_3 \}$ respectively from each disjunctive constraint 1-6 above. The second solution is to first buy the milk, return home, and then go again to the store and buy the fruit represented by the disjuncts $\{ s_2 \rightarrow s_6 , s_3 \rightarrow s_6 , s_3 \rightarrow s_6 , s_3 \rightarrow s_4 , s_3 \rightarrow s_4 , s_3 \rightarrow s_4 \}$. Actually, the first solution is implied the conjunction of $T_1 \cup T_2$ and by just selecting $s_6 \rightarrow s_1$ and the second by $T_1 \cup T_2$ and selecting $s_3 \rightarrow s_4$ (in other words the STP $T_1 \cup T_2 \cup \{ s_6 \rightarrow s_1 \}$ subsumes all constraints 1-6). The third solution is to merge the returning to home steps and select the disjuncts $\{ s_2 \rightarrow s_6 , Merge(s_3 , s_6) , Merge(s_3 , s_6) , s_5 \rightarrow s_3 , Merge(s_3 , s_6) , Merge(s_3 , s_6) \}$, implied by simply imposing $Merge(s_3 , s_6)$[29] in conjunction with $T_1 \cup T_2$.

Because of the way in which the plan-merging problem is defined, any DTP subset of the DTP defined by the constraints 1-6 above (i.e. a DTP containing only a subset of disjuncts per disjunction) that contains at least one consistent component STP (i.e. a solution) is a solution to the plan-merging problem. For example, selecting any of the three STPs corresponding to the solutions above is a solution T' to the plan-merging problem. The question that arises is which of all possible solutions T' to the merging problem is preferable?

If an agent iteratively solves plan-merging problems adopting and committing to new plans in the midst of executing others, and desires to be complete (i.e. always find a solution to the plan-merging problem if there is one), then *the agent should adopt the whole DTP defined by the CRC constraints as the plan-merging solution* (or any equivalent subset). We will show this with an example. Suppose the agent has successfully merged P_1 and P_2 above to a new plan P_3 by selecting as the merging solution the STP component of CRC_{12} (let us call it T') in which she first buys the milk and then the fruit. The constraints of T' are now part of the constraints T of plan P_3. Now suppose that she

[29] The reader might notice that in the third solution, only the steps for returning home need to be merged, which is semantically wrong: the steps for going to the store should also be merged. This is because we did not attach any preconditions and causal-links to the *Go-From-Home-To-Store* steps. We will fix the problem below when we discuss the initial state problem.

Plan-Merge1 ($P_1 = <S_1 , T_1 , L_1 >$, $P_2 = <S_2 , T_2 , L_2 >$)

1. Let $P' = <S_1 \cup S_2 , T_1 \cup T_2, L_1 \cup L_2 >$
2. Identify all conflicts in P' by checking for all pairs of steps in $S_1 \cup S_2$ whether the conditions in Definition 7-1, Definition 7-2, or Definition 7-3 are satisfied[30].
3. If $T_1 \cup T_2 \cup CRC_{12}$ has a solution
4. Return $P'' = <S_1 \cup S_2 , T_1 \cup T_2 \cup CRC_{12}, L_1 \cup L_2 >$
5. Else
6. Return *Inconsistent-Plans*

Figure 7-3: The Plan-Merge1 algorithm for propositional, non-conditional plans.

tries to merge a plan P_4 that creates new conflicts and which are impossible to resolve if we impose the constraints implied by first buying the milk, i.e. T'. Since however, the agent has threw away the previous choices for resolving the conflicts between P_1 and P_2 , it has committed to buying the milk first and thus cannot find a solution to the merging problem.

The fact that the whole *CRC* should be returned as the plan-merging solution does not necessarily imply that the agent is required to execute the resulting plan as a DTP, even though we have provided an algorithm for this in Chapter 5. She could select any component STP of $T_1 \cup T_2 \cup CRC_{12}$ and execute that instead (e.g. with the algorithm in [Muscettola, Morris et al. 1998]). Also notice that the *CRC* could be simplified by throwing away any disjuncts that are inconsistent with $T_1 \cup T_2$ and thus never have the chance to participate in any solution of the current or future plan-merging problem. For example, there is no solution in the example above that involves the introduction of white knights since $(s_6 \rightarrow s_4 \rightarrow s_2)$, $(s_6 \rightarrow s_4 \rightarrow s_3)$, $(s_3 \rightarrow s_1 \rightarrow s_5)$, $(s_3 \rightarrow s_1 \rightarrow s_6)$ are inconsistent with the constraints T_1 and T_2 in plans P_1 and P_2 respectively; thus these disjuncts can be safely removed from CRC_{12}.

The above discussion gives rise to the plan-merging algorithm in Figure 7-3. Plan P' is the plan resulting from considering two plans P_1 and P_2 together, while plan P'' is the same plan with additional conflict constraints, any solution of which is a conflict free plan.

The problem with the initial state

In classical partial-order planning a typical trick used by the algorithms is to insert two dummy steps $s_{initial}$ and s_{goal} in the empty plan before starting to search for a plan that achieves the goals. Step $s_{initial}$ has no preconditions and its effects are the predicates known to be true at the initial state. Step s_{goal} has no effects and its preconditions are all the goal predicates that have to be achieved. All steps that are later added in the plan are constrained to occur after $s_{initial}$ and before s_{goal} . One of the reasons for adding these two dummy steps is uniformity. For example, when a precondition p of a step s is satisfied by the initial step, the causal-link $< s_{initial}, p , s>$ is added to the plan. Therefore, the situation of a precondition being satisfied by another step, or being already satisfied by the initial state is handled in a uniform way.

[30] Actually, it is not necessary to check whether the conditions that involve temporal constraints in the definitions hold (e.g. conditions 3 and 4 in Definition 7-1). Even if they do not hold and we generate non-existing conflicts, the DTP solver will be

In plan-merging we have to take special care of the initial state. For example, suppose that we have to merge plans P_1 and P_2 , where P_1 represents the conjunction of the agent's existing plans, and P_2 will represent its plan for a new goal. Plan P_1 thus already contains a dummy step $s_{initial}$ while plan P_2 is a freshly instantiated plan from the plan schemata library. The library plans should not contain dummy steps $s_{initial}$ because only one such step can correspond to the initial state. Thus, we have no way to connect the new plan to the initial state.

For an agent architecture based on plan-merging, we suggest the following solution to the above problem. The plan schemata could be defined in such a way that any instantiations of them can be applied in any initial state and there is no step precondition that should be satisfied by the initial state. This has been the case in the applications that we have been looking at such as the Plan Management Agent [Tsamardinos, Pollack et al. 1999]. If this restriction is impossible to satisfy in some domain, then plan schemata can have a set of step predicates (we will call them ***plan conditions***) that have to be satisfied by the initial state. Then, when a plan-schema is instantiated the plan conditions are checked. If they are not met by the initial state the plan cannot be merged. If they are, then before we attempt merging the plan we add the causal links $<s_{initial}, p, s>$ for every plan condition p , where s is the step that has as precondition the predicate p.

Again, the latter solution to the problem is restrictive in some ways. For example, it might be impossible to merge plan P_2 with the rest of the commitments P_1 if we add the causal-links that we suggest, but it might be possible to merge P_2 if the plan conditions are satisfied some other way, e.g. by other steps in P_1. Nevertheless, this case is out of the scope of this section and requires advanced techniques such as plan repair or plan-generation in the general case.

Let us examine how we could apply the above suggestion to the merging problem of Example 7-1. Let us suppose, as semantically correct, that steps *Go-From-Home-To-Store* (s_1 and s_4) have preconditions *At-Home* and ¬*At-Store* . Thus, both plans P_1 and P_2 will have as plan conditions {*At-Home*, ¬*At-Store*}. We will identify the initial step $s_{initial}$ with *TR* (represented by only time-point in the figure for simplicity and since, as a dummy step, it does not have temporal extent). Then, if at time *TR* these preconditions are satisfied, we can attempt to merge the two plans and add the causal links {<*TR, At-Home,* s_1 >, {<*TR, At-Home,* s_4 >, {<*TR,* ¬*At-Store,* s_1 >, {<*TR,* ¬*At-Store,* s_4 > to plan *P'* of algorithm **Plan-Merge1** before in line 2 in Figure 7-3 we identify the conflicts. The result is shown in Figure 7-4.

able to trivially resolve these phantom conflicts in the *CRC*: since the temporal constraints in the definitions do not hold, their negation holds and thus their conflict-resolution constraints trivially hold.

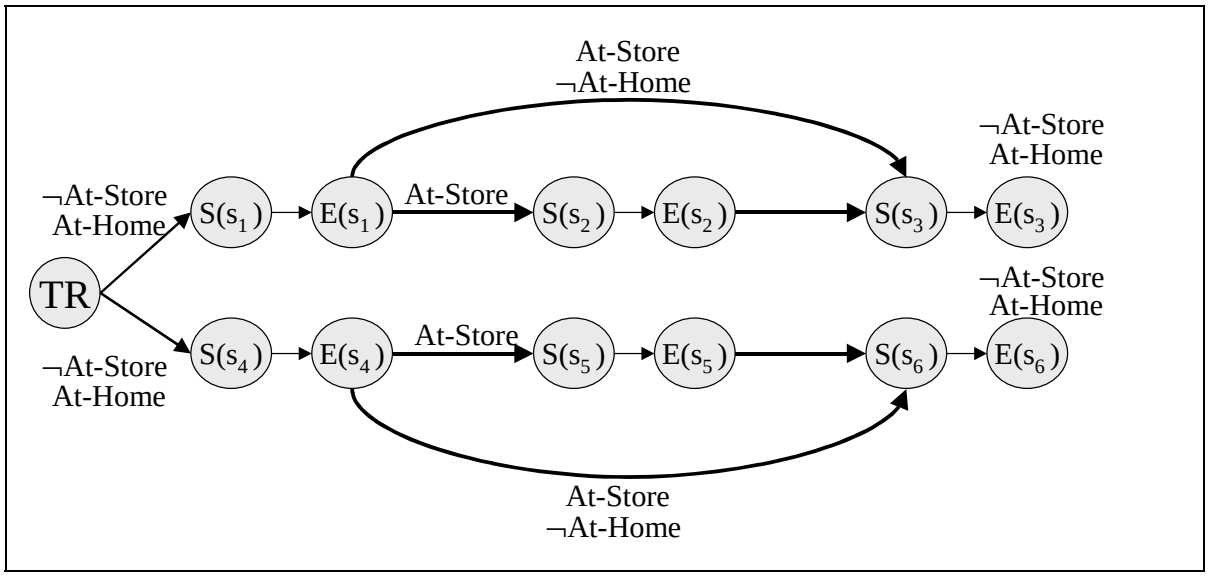


Figure 7-4: The plans of Example 7-1 with additional causal-links from the initial state.

The new causal-links that we introduced cause additional conflicts:

1. s_4 conflicts with *TR* because of $<TR, At\text{-}home, s_1>$
2. s_4 conflicts with *TR* because of $<TR, \neg At\text{-}Store, s_1>$
3. s_1 conflicts with *TR* because of $<TR, At\text{-}home, s_4>$
4. s_1 conflicts with *TR* because of $<TR, \neg At\text{-}Store, s_4>$

The conflict resolution constraints for these particular conflicts are:

1. $\{s_4 \rightarrow TR\} \vee \{s_1 \rightarrow s_4\} \vee Merge(s_1, s_4) \vee \{s_4 \rightarrow s_6 \rightarrow s_1\} \vee \{s_4 \rightarrow s_3 \rightarrow s_1\}$
2. $\{s_4 \rightarrow TR\} \vee \{s_1 \rightarrow s_4\} \vee Merge(s_1, s_4) \vee \{s_4 \rightarrow s_6 \rightarrow s_1\} \vee \{s_4 \rightarrow s_3 \rightarrow s_1\}$
3. $\{s_1 \rightarrow TR\} \vee \{s_4 \rightarrow s_1\} \vee Merge(s_1, s_4) \vee \{s_1 \rightarrow s_6 \rightarrow s_4\} \vee \{s_1 \rightarrow s_3 \rightarrow s_4\}$
4. $\{s_1 \rightarrow TR\} \vee \{s_4 \rightarrow s_1\} \vee Merge(s_1, s_4) \vee \{s_1 \rightarrow s_6 \rightarrow s_4\} \vee \{s_1 \rightarrow s_3 \rightarrow s_4\}$

The previous solutions to this merging problem, i.e. before the addition of the causal-links above and the introduction of the new conflicts, can be extended to resolve these conflicts too. For example, if we decide to buy the fruit first, then s_1 is ordered after s_4 and s_6 which resolves conflicts 1 and 2 by satisfying the third disjunct (and so s_6 acts a white-knight), and conflicts 3 and 4 are satisfied by promotion, i.e. $s_4 \rightarrow s_1$. Similarly, if we decide to buy the milk first, we can resolve conflicts 1-4. If we decide to merge the steps to return to home s_3 and s_4 , then the only way left to resolve the conflicts 1-4 is by selecting to merge s_1 and s_4 . The reader should check that actually there are no other solutions and that the new constraints do not increase the number of solutions.

Plan-Merging architectures define Dynamic DTP problems

Let us consider two consecutive plan merging problems. In the first one we merge $P_1 = <S_1, T_1, L_1>$ with $P_2 = <S_2, T_2, L_2>$. The algorithm in Figure 7-3 checks whether the DTP defined by the constraints $DTP_1 = T_1 \cup T_2 \cup CRC_{12}$ has a solution and if yes, it merges the two plans

producing $P_3 = <S_1 \cup S_2 , T_1 \cup T_2 \cup CRC_{12}, L_1 \cup L_2 >$. Next suppose that we merge P_3 with $P_4 = <S_1 , T_1 , L_1 >$. Again, algorithm **Plan-Merge1** will invoke a DTP solver to determine whether $DTP_2 = T_1 \cup T_2 \cup CRC_{12} \cup CRC_{34}$ has a solution. This example illustrates that in consecutive plan-merging problems, where new plans are continuously being merged with the previously adopted plans, Dynamic DTPs arise. In this case DTP_2 is the same as DTP_1 with the addition of the constraints CRC_{34} and the new time-points that appear in latter set of constraints.

Let us denote as $ngs(DTP_i)$ the no-goods recorded when solving DTP_i by Epilitis. Once we have already solved DTP_1 we can speed up solving DTP_2 by using $ngs(DTP_1)$. Since, DTP_2 only contains additional constraints and no constraint has been relaxed, all no-goods in DTP_1 are still applicable in DTP_2 and can be used to prune the search space.

7.3 Extending the Plan Merging Algorithm to Include Conditional Plans

In Chapter 6 we developed the class of problems called CDTPs that can be used to represent conditional plans [Peot and Smith 1992] with quantitative temporal constraints. In this section we show how to use this formalism to extend the plan-merging algorithm of the previous section to such plans.

Let us represent with $V(S)$ the temporal variables defined by *Start(s)* and *End(s)* for all steps s in S union the time reference point TR. A conditional plan P can be represented by a tuple $<S, T, L, Labels, OV, O, Prop >$, where S, T, and L are as before the set of steps, temporal constraints, and causal-links respectively, and $Prop$ is a finite set of timed propositions (with $Prop^*$ being the label universe, see Definition 6-5), $Labels$ is a function $V(S) \rightarrow Prop^*$ attaching a label to each node, OV is the set of observation nodes, and O is a 1-1 and "onto" function $Prop \rightarrow OV$ associating an observation variable with a proposition, just as in Definition 6-6 for the CSTP. Given a plan $P = <S, T, L, Labels, OV, O, Prop >$, the tuple $<V(S), T, Labels, OV, O, Prop>$ defines a CDTP.

Extending algorithm **Plan-Merging1** for conditional plans is relatively straightforward once we have modified the definitions of the conflicts accordingly. For example, a step s_t does not clobber a causal link $<s_p , e , s_c >$ if s_t and the causal link never appear together in any scenario. We now repeat the definitions of the conflicts to take into account the fact that steps have a label attached and will only be executed under certain conditions. The differences from the definitions for the non-conditional case are in italics.

Definition 7-6: A ***Causal-Conflict*** between steps s_p and s_t in *conditional* plan $P = <S, T, L, Labels, OV, O, Prop >$ exists if the following conditions hold:

1. $<s_p , e , s_c> \in L$, for some other step s_c in S.
2. There is some other step s_t that has an effect $\neg e$ and *there is at least one scenario sc in which steps* s_p , s_t , s_c *are all executed together.*
3. Step s_t might begin before step s_c ends, i.e. $Start(s_t) - End(s_c) \leq 0$ is consistent with $Pr_T(sc)$ (recall that $Pr_T(sc)$ is the projection STP or DTP of the CSTP or DCTP respectively to scenario *sc*).
4. Step s_t might end after step s_p ends, i.e. $Start(s_p) - End(s_t) \leq 0$ is consistent with $Pr_T(sc)$.

5. *For any scenario sc in which* s_p , s_t , s_c *all appear together* , there is no other step s_k (executed in the *sc*) with effect *e*, that necessarily comes between s_t and s_c , i.e. $s_t \rightarrow s_k \rightarrow s_c$ in $Pr_T(sc)$.

Definition 7-7: An ***In-Condition-Conflict*** between two different steps s_p and s_t in *conditional* plan $P = <S, T, L, Labels, OV, O, Prop>$ exists if the following conditions hold:

1. Step s_p has an in-condition *i*.
2. Step s_t has an in-condition $\neg i$.
3. *There is a scenario sc in which* steps s_p and s_t *are both executed* and they might overlap, i.e. $Start(s_t) - End(s_p) \leq 0$ and $Start(s_p) - End(s_t) \leq 0$ are both consistent with $Pr_T(sc)$.

Definition 7-8: A ***Recourse-Conflict*** between two different steps s_p and s_t in *conditional* plan $P = <S, T, L, Labels, OV, O, Prop>$ exists if the following conditions hold:

1. Steps s_p and s_t both have the same resource *r*.
2. *There is a scenario sc in which* steps s_p and s_t *are both executed* and they might overlap, i.e. $Start(s_t) - End(s_p) \leq 0$ and $Start(s_p) - End(s_c) \leq 0$ are both consistent with $Pr_T(sc)$.

The definition of the plan-merging problem also has to be modified:

Definition 7-9: Let $P_1 = <S_1 , T_1 , L_1 , Labels_1 , OV_1, O_1, Prop_1 >$ and $P_2 = <S_2 , T_2 , L_2 \; Labels_2 , OV_2, O_2, Prop_2 >$. Consider the plan $P' = <S_1 \cup S_2 , T_1 \cup T_2 \cup T', L_1 \cup L_2 , Labels_1 \cup Labels_2 , OV_1 \cup OV_2 , O_3 , Prop_1 \cup Prop_2>$, where O_3 is the function $Prop_1 \cup Prop_2 \rightarrow OV_1 \cup OV_2$ associating the observations to an observation node in the plan. The ***plan-merging*** problem is to find a set of temporal constraints T' such that the plan $P'' = <S_1 \cup S_2 , T_1 \cup T_2 \cup T', L_1 \cup L_2 , Labels_1 \cup Labels_2 , OV_1 \cup OV_2 , O_3 , Prop_1 \cup Prop_2>$ can be executed in a conflict-free way, i.e. the CDTP defined by the constraints $T_1 \cup T_2 \cup T'$ for plan P' is either Strong, Weak, Dynamic Consistency (depending on the required semantics) and it resolves all conflicts.

Plan-Merge2 ($P_1 = <S_1 , T_1 , L_1 , Labels_1 , OV_1, O_1, Prop_1 >$,
$P_2 = <S_2 , T_2 , L_2 , Labels_2 , OV_2, O_2, Prop_2 >$,
Consistency notion *Concistency-notion*)

1. Let $P' = <S_1 \cup S_2 , T_1 \cup T_2 \cup T', L_1 \cup L_2 , Labels_1 \cup Labels_2 , OV_1 \cup OV_2 , O_3 , Prop_1 \cup Prop_2>$, where O_3 is the function $Prop_1 \cup Prop_2 \rightarrow OV_1 \cup OV_2$ associating the observations to an observation node in the plan.
2. Identify all conflicts in P' by checking for all pairs of steps in $S_1 \cup S_2$ whether the conditions in Definition 7-6, Definition 7-7, or Definition 7-8 are satisfied.
3. If CDTP with constraints $T_1 \cup T_2 \cup CRC_{12}$ is consistent using *Consistency-notion*
4. Return $P'' = <S_1 \cup S_2 , T_1 \cup T_2 \cup CRC_{12}, L_1 \cup L_2 , Labels_1 \cup Labels_2 , OV_1 \cup OV_2, O_3 , Prop_1 \cup Prop_2>$
5. Else
6. Return *Inconsistent-Plans*

Figure 7-5: The Plan-Merge2 algorithm for propositional, conditional plans.

A technical detail is that the sets of observations should be disjoint. The above definition of plan-merging, although formally more complex, is intuitively the same as for the non-conditional case.

In Chapter 6 we used the notation $N(v, sc)$ to denote the node v in the projection STP of scenario sc . We will adopt the same notation now and use the notation $N(Start(s), sc))$ to denote the start time-point of step s in scenario sc and the notation $N(s_1, sc) \rightarrow N(s_2, sc)$ to denote the ordering constraint specifying that the end of step s is before the start of step s in scenario sc. Also, we extend the notation $Merge(s_t , s_c)$ to take as additional argument a scenario sc in which the nodes should be merged, i.e. $Merge(s_t , s_c , sc)$ to denote the constraint $N(Start(s_1), sc) = N(Start(s_2), sc) \wedge N(End(s_1), sc) = N(End(s_2), sc)$. We are now ready to extend the conflict-resolution constraints to apply to the definitions of conflicts in the conditional case:

Conflict-Resolution Constraint for causal-conflict between s_p, s_t and causal link : $<s_p , e , s_c>$

For any scenario sc consistent with all s_t , s_p , s_c return the conjunction of the constraints:

$$N(s_c , sc) \rightarrow N(s_t , sc) \vee$$
$$N(s_t , sc) \rightarrow N(s_p , sc) \vee$$
$$(Mergable(s_t , s_i) \wedge Merge(s_t , s_c , sc)) \vee$$
$$\vee_{s \in S_e \wedge Con(s,sc)} N(s_t, sc) \rightarrow N(s, sc) \wedge N(s, sc) \rightarrow N(s_c, sc)$$

where again S_e is the set of steps with effect e and $Con(s, sc)$ is the predicate that denotes whether the two labels s and sc are consistent.

Conflict-Resolution Constraint for in-condition-conflict between s_p , s_r and in-condition *i*:

For any scenario sc consistent with all s_t , s_p , s_c return the conjunction of the constraints:

$$N(s_p , sc) \rightarrow N(s_t , sc) \vee$$
$$N(s_t , sc) \rightarrow N(s_p , sc)$$

Conflict-Resolution Constraint for resource-conflict between s_p, s_t and resource *r*:

For any scenario sc consistent with all s_t , s_p , s_c return the conjunction of the constraints:

$$N(s_p , sc) \rightarrow N(s_t , sc) \vee$$
$$N(s_t , sc) \rightarrow N(s_p , sc) \vee$$
$$(Mergable(s_p , s_t) \wedge Merge(s_p , s_t , sc))$$

Given the above definitions the algorithm **Plan-Merge1** should be modified to the one shown in Figure 7-5.

7.4 Extending the Plan Merging Algorithm to Include n-ary Predicates

It is typically the case that plan representation languages allow predicates and actions to contain variables. For Example 7-1 this feature would enable us to dispense with the steps *Go-From-Home-to-Store* and *Go-From-Store-to-Home* and instead define the operator *Go(from, to)* , where *from* and *to* are variables, whose instantiations *Go(Home, Store)* and *Go(Store, Home)* would correspond to the steps s_1 , s_3 , s_4, and s_6 in the example. We are using the convention that variables start with a lower case letter and constants with an upper-case letter. The preconditions and effects of step *Go-From-Home-to-Store* are {*At-Home*} and {¬*At-Home, At-Store*} respectively. Similarly, the preconditions and effects of *Go-From-Store-to-Home* are {*At-Store*} and {*At-Home*, ¬*At-Store*}. Instead, if we allow

variables in the predicates in the preconditions and effects operator *Go(from, to)* will have preconditions {*At(from)* } and effects {¬*At(from), At(to)* }.

In the first approach in which there are no variables, if there are 50 possible places the agent might be, we would have to define 50 × 50 = 2500 types of steps with appropriate effects and preconditions for going from one place to the other. Instead, when using variables, we would just define the operator *Go(from, to)* and only one set of preconditions and effects. The argument for using this abstract description is that it has software engineering benefits: needless duplication would likely lead to inconsistent domain definitions if an error in one copy of a step was replaced by other copies were mistakenly left unchanged [Weld 1994]. Indeed many systems use representations that employ variables for the operators and n-ary predicates [Penberthy and Weld 1992; Ghallab and Laruelle 1994; Muscettola 1994; Onder and Pollack 1999] to name but a few.

For the rest of this section we will assume that the steps in the plans are instances of operators and may contain variables and n-ary predicates. For example, the block's world the operator *move* is defined as:

Operator *move(b, x, y)*

Preconditions:	*on(b, x)* ∧ *clear(b)* ∧ *clear(y)*
Effects:	*on(b, y)* ∧ ¬*on(b, x)* ∧ *clear(x)* ∧ ¬*clear(y)*

When we need to instantiate the above operator, we might specify that $b = A$ and $x = B$ where A and B are specific objects in the domain. This particular instantiation *move(A, B, y)* represents moving block A from block B to some other block y , where y is still an unbound variable. Apart from value binding constraints of the form *var = Object* , we will allow codesignation and non-codesignation constraints of the form *var1 = var2* and *var3 ≠ var4* respectively. For example, if we have two steps *move(A, B, y)* and *move(B, C, z)* and the constraint $y=z$ in a plan, then the plan specifies that A should be moved from B to y , and B from C to the same block y (=z). We will use ***the set B of binding constraints*** in a plan to store the codesignation, non-codesignation, and value binding constraints.

Example 7-2: We will now encode the plan in Example 7-1 using operators and predicates with variables. We will define the operators:

Operator *Go(from, to)*:

Preconditions:	*At(from)*
Effects:	*At(to)*

Operator *Buy(good)*

Preconditions:	*At(Store)*
Effects:	*Have(good)*

Then plan P_1 and P_2 are defined as:

- S_1 = {s_1 = *Go(Home, Store)*, s_2 = *Buy(Milk)* , s_3 = *Go(Store, Home)*}
- T_1 are the ordering constraints $s_1 \rightarrow s_2 \rightarrow s_3$
- L_1 = {<TR, *At(Home)*, s_1> , <s_1 , *At(Store)* , s_2 > , <s_1 , *At(Store)*, s_3 > }
- B_1 are the variable binding constraints implied by the way we denote the steps in S_1
- S_2 = {s_4 = *Go(Home, Store)*,, s_5 = *Buy(Fruit)* , s_6 = *Go(Store, Home)*}

- T_2 are the ordering constraints $s_1 \rightarrow s_2 \rightarrow s_3$
- $L_2 = \{<TR, At(Home), s_4> , <s_4 , At(Store) , s_5 > , <s_4 , At(Store), s_6 > \}$
- B_2 are the variable binding constraints implied by the way we denote the steps in S_2

In addition, we will define predicate *At* to be a functional predicate, i.e. $At(x) \wedge x \neq y \Rightarrow \neg At(y)$. Plans P_1 and P_2 are shown in Figure 7-6.

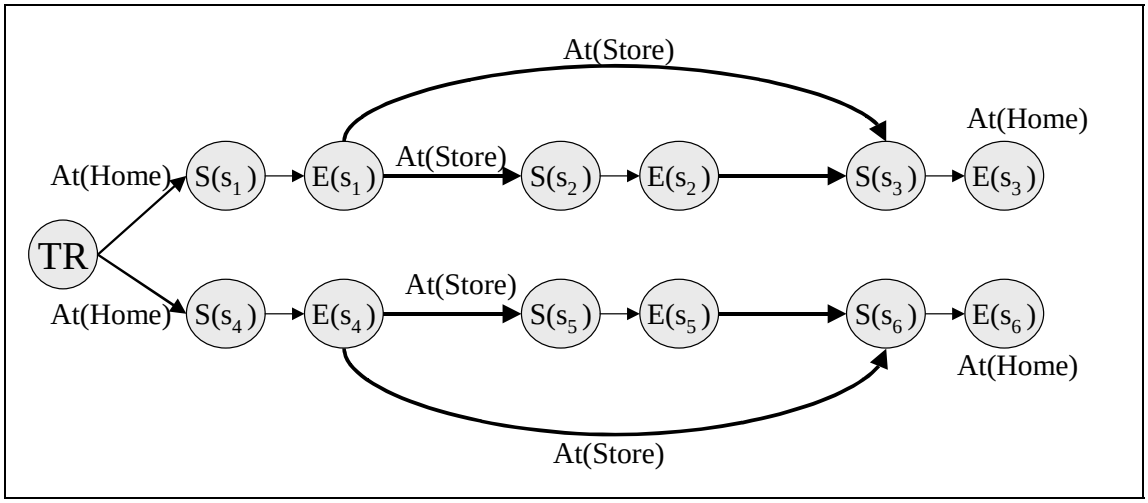


Figure 7-6: The plans of Example 7-1 using a representation that allows n-ary predicates and variables.

Allowing variables and n-ary predicates in the representation requires different definitions for the conflicts and appropriate changes to the conflict-resolution constraints. The rest of the section is an informal discussion on how to perform plan-merging using this representational approach. For a formal definition of the conflicts see [Yang 1992; Yang 1997].

Consider for two steps s_1 and s_2 in two different plans that are being merged and that both use a printer as a resource, i.e. *Printer(x)* and *Printer(y)* respectively. The steps are allowed to overlap by the union of the temporal constraints in the two plans. The question is, is there a conflict between these two steps? If x and y are bound to the same printer, then obviously there is a conflict between the two steps because they might attempt to use the same printer at the same time.

However, at merging time, the steps might only be partially instantiated, meaning that some or all variables might be left unbound. Let us suppose that there are three printers *A, B,* and *C* in the domain and we are trying to resolve the potential conflict between s_1 and s_2 . As before, we can impose ordering constraints between them so that they do not overlap. In addition, we can impose codesignation, non-codesignation, or value binding constraints (that we will call ***binding constraints***) to ensure that they do not use the same printer:

$$s_1 \rightarrow s_2 \vee s_2 \rightarrow s_1 \vee (x = A \wedge y = B) \vee (x = A \wedge y = C) \vee (x = B \wedge y = C) \vee (x = C \wedge y = A) \vee (x = C \wedge y = B)$$

or similarly,

$$s_1 \rightarrow s_2 \vee s_2 \rightarrow s_1 \vee x \neq y$$

The first constraint corresponds to an *eager-commitment* approach in which bindings for variables are chosen as soon as possible. The second constraint (disjunction) corresponds to a *delayed* or *least-*

commitment approach. The idea in the latter approach is to represent possible bindings of a variable in a plan *collectively*, and to instantiate a variable to a constant value only when necessary, e.g. at execution time. As mentioned in [Yang 1997], this approach to reasoning about resources has been adopted by a number of practical planning systems, such as MOLGEN [Stefik 1981] , SIPE [Wilkins 1988], GEMPLAN [Lansky 1988], and scheduling systems such as ISIS [Fox 1987] and HSTS [Muscettola 1994], and so we will adopt it for the discussion in this section.

In the previous example, if the predicates representing the resources are n-ary, then they will create n number of disjuncts in the conflict-resolution constraint. For example, if steps s_1 and s_2 of the previous example also have another resource $r(m, n)$ and $r(p, q)$ respectively, then we should post the following constraint to resolve the potential conflict:

$$s_1 \rightarrow s_2 \vee s_2 \rightarrow s_1 \vee m \neq n \vee p \neq q$$

Finally, if the predicates are functional, e.g. $At(x)$, then this fact has to be taken into consideration. For example, suppose that steps s_1 and s_2 have the in-condition $At(x)$ and $At(y)$ and they might overlap. If we ignore the fact that $At(x)$ is functional then there is no conflict between the steps. This implies that predicates $At(x)$ and $At(y)$ could hold at the same time, even if $x \neq y$. For a non-functional predicate this semantics would be correct: e.g. for predicate $Have(x)$, we should allow $Have(Milk)$ and $Have(Fruit)$ to hold simultaneously, which is not true for predicate At. Thus, for such a functional predicate a conflict exists and we should post the constraints:

$$s_1 \rightarrow s_2 \vee s_2 \rightarrow s_1$$

and not allow the two steps to overlap.

In the presence of n-ary predicates, the plan-merging problem is redefined to not only finding a set of temporal constraints that resolves the conflicts, but also a set of binding constraints. However, since now the set of constraints includes non-temporal constraints the plan-merging problem cannot be solved with Epilitis as it is currently implemented. Fortunately, extending Epilitis to take into consideration binding constraints is feasible and the topic of the next subsection.

7.4.1 Extending Epilitis to handle binding constraints

To extend the no-good learning algorithm in Chapter 3 (resulting to algorithm Epilitis) to be able to handle temporal constraints we required or modified the following set of functions:

Maintain-consistency: This function propagates a temporal constraint maintaining the current STP in path-consistency.

Forward-check: This function removes all values c_{ij} that are inconsistent with the current STP by employing the FC-Condition and the fact that the current STP is path-consistent.

Justification-value This function returns a minimal justification why a value is inconsistent with the current STP.

Equivalently, in order to handle binding constraints in Epilitis we will need to modify the above functions:

Maintain-consistency This function should propagate c_{ij} and maintain the current STP in path-consistency as before, if c_{ij} is a temporal constraint. In addition,

	it should propagate a binding constraint c_{ij} in some other data structure, so to facilitate forward-checking binding constraints.
Forward-check:	This function should remove all values c_{ij} that are inconsistent with the current STP (if c_{ij} is a temporal constraint $x - y \leq b$), or inconsistent with the current set of binding constraints (if c_{ij} is a binding constraint $x = y$, $x \neq y$, or $x = Object$).
Justification-value	This function should work as before for temporal constraints c_{ij}. For binding constraints c_{ij} it should again return a minimal justification why a binding constraint is inconsistent with the current set of binding constraints.

Epilitis extended to handle binding constraints as described above would return a set of consistent temporal and binding constraints. However, this is not exactly the solution to the plan-merging problem. Consider the set of binding constraints $x \neq y$, $y \neq w$, $w \neq x$. The three constraints appear consistent, but if there are only two objects that we can assign to x, y, and w, there is no assignment to the variables that satisfies all constraints. Thus, with the above extensions we just described Epilitis returns a candidate solution; the binding constraints of that candidate solution should also be checked with a standard CSP solver and make sure there is an assignment of objects to the predicate variables that satisfies the binding constraints. In case there is no such assignment, Epilitis should backtrack and look for another candidate solution. In the case where the CSP solver is able to additionally return a minimal justification for a failure, Epilitis could use information for Conflict-Directed Backjumping and no-good learning.

We will defer further discussion, implementation, and experimentation of such an extension for future work. Yang in [Yang 1997] describes a solver that can handle qualitative temporal constraints and binding constraints and his approach could directly be adopted for handling quantitative temporal constraints and binding constraints. Nevertheless, he does not use all the pruning techniques in Epilitis palette and we believe there is much room for improvement in his approach.

7.5 Step Merging, Step Reuse, Cost-in-Context, and Cost Minimization

In Section 7.2.3 we defined step-merging as a way of resolving conflicts. It is also possible however, to use step-merging as a way to reduce the cost of a plan. Let us assume for example, that the steps to go to the store and return home in Example 7-1 have a cost associated with them, e.g. the price of the gas to drive between these places. When we consider the two plans P_1 and P_2 together, we would obviously prefer a solution to the plan-merging problem in which we merge the *Go-From-Home-to-Store* and *Go-From-Store-to-Home* steps, rather than first going to the store, buying the milk, and then returning home.

Step-merging and plan cost-minimization is important for many domains. Yang for example mentions that identical plan-merging issues arise in the domain of automated manufacturing where process plans for metal-cutting [Raghu Karinthi, Nau et al. 1992], set-up operations [Hayes 1989], and tool-approach directions [Mantyla and Opas 1988] need to be optimized; similar arguments

hold in the area of query optimization [Sellis 1988] in database systems, as well as domains having multiple agents [Durfee and Lesser 1987; Rosenblitt 1991].

In general, it is possible that we can substitute any set of steps S_1 with another set of steps S_2 so that S_2 collectively has the same preconditions and effects as S_1 still has less total cost than the first one; or S_2 has greater cost than S_1 but the substitution allows the merging of other steps reducing the total cost. Here however, we only consider substituting a pair of steps of identical type with a single step of the same type. In addition, we only consider additive cost-functions in which the total cost is the sum of the cost of the individual steps. For clarity of presentation we will only consider propositional plans with no conditions. The ideas in Section 7.3 and Section 7.4 can be used to extend the algorithms that we are about to present to conditional and non-propositional plans.

Definition 7-10: Consider a plan $P = <S, T, L>$ where each step $s \in S$ has a cost $c(s)$ associated with it and some pairs of steps are mergable with each other as denoted by predicate $Mergable(s_1 , s_2)$. The ***step-merging*** problem is to find a set of step-merges $Merge(s_1 , s_2)$ so that the set of temporal constraints M that the merges imply are consistent with T and the sum of the costs of the steps is minimized.

Consider a pair of steps s_1 and s_2 in the plan that are mergable. Deciding to merge them imposes additional temporal constraints to the plan that can potentially render the plan inconsistent or remove the possibility of merging some other pair of steps. Thus, we have to collectively determine the consistency of the merging decisions we take. We now present an algorithm for solving the step-merging problem as defined above by transforming the problem to a DTP, shown in Figure 7-1. The constraint {} at line 3 is the empty constraint that is always satisfied. To express {} in DTP format we can write $x - y \leq \infty$ for any DTP variables x and y. Notice that the constraints M that we add to T do not remove any solutions from DTP T since the agent may choose to merge nothing by selecting all disjuncts that correspond to empty {} constraints.

Step-Merge(*P = <S, T, L>*)
1. $M = \varnothing$
2. For every pair of steps s_1 , s_2 for which $Mergable(s_1 , s_2)$ is *True*
3. $\quad M = M \cup \{ Merge(s_1 , s_2) \vee \{\}\}$
4. Find all solutions to DTP $T \cup M$ and return the one with minimum cost

Figure 7-7: Algorithm Step-Merge

7.5.1 Cost in context evaluation

The step-merge algorithm for cost minimization that we presented in the previous section can be used in conjunction with the plan-merging algorithm in Section 7.2. That is, when we merge two plans together we might require that we not only we resolve all the conflicts that arise, but also perform step-merges to minimize the total cost. The algorithm that finds a solution to the two problems simultaneously is presented at Figure 7-8.

Minimum-Cost-Merge($P_1 = <S_1 , T_1 , L_1 >$, $P_2 = <S_2 , T_2 , L_2 >$)

1. Let $P' = <S_1 \cup S_2 , T_1 \cup T_2, L_1 \cup L_2 >$
2. Identify all conflicts in P' by checking for all pairs of steps in $S_1 \cup S_2$ whether the conditions in Definition 7-1, Definition 7-2, or Definition 7-3 are satisfied.
3. $M = \varnothing$
2. For every pair of steps s_1 , s_2 in P' for which *Mergable*(s_1 , s_2) is *True*
3. $\quad M = M \cup \{ Merge(s_1 , s_2) \vee \{\}\}$
4. Find the minimum cost solution to DTP $T_1 \cup T_2 \cup CRC_{12} \cup M$ and assign it to *Min-Cost-Solution*
5. If $T_1 \cup T_2 \cup CRC_{12} \cup M$ has at least one solution
6. $\quad$ Return $< P'' = <S_1 \cup S_2 , T_1 \cup T_2 \cup CRC_{12} \cup M, L_1 \cup L_2 >$, *Min-Cost-Solution*>
7. Else
8. $\quad$ Return *Inconsistent-Plans*

Figure 7-8: Algorithm for Minimum-Cost-Merge

The algorithm **Minimum-Cost-Merge** takes as input two plans and returns two objects: (i) a plan that is the merge of the two plans with additional constraints so that any solution of the constraints resolves all the conflicts between the plans, and (ii) it returns the solution (i.e. the STP component) with the minimum cost.

The algorithm can be used to solve the cost-in-context problem [Horty and Pollack 1998; Horty and Pollack 2001] that we are about to present. Suppose that an agent has committed to the set of plans P_1 and considers adopting a new plan P_2 with utility $Utility(P_2)$ and cost $Cost(P_2)$. The cost of executing P_2 in isolation is $Cost(P_2)$ but in the context of the adopted plan P_1 it might be less than that; e.g. if the agent has adopted the plan to go to the store and buy milk, then the cost of a plan to go to the store and buy fruit *in the context of the first plan* is less because the agent can merge the steps of going and coming from the store. We denote $Cost(P_2 \mid P_1)$ the minimum cost of P_2 in the context of P_1 , i.e. given the fact that we have already committed to P_1 . The problem then is whether the agent should adopt and commit to plan P_2 or not.

The answer is that the agent should commit to plan P_2 if the cost in the context of P_1 is lower than the utility of plan P_2. Here we assume that the utility of P_2 is independent of the context, otherwise we would also have to calculate $Utility(P_2 \mid P_1)$. Algorithm **Minimum-Cost-Merge** returns $Cost(P_2 \mid P_1)$ as the second returned value which can be used to determine whether:

$$Utility(P_2) - Cost(P_2 \mid P_1) > 0$$

in which case the agent should adopt P_2 , or not, in which case the agent should not adopt P_1.

7.6 Related Work, Discussion, Contributions, and Future Work

7.6.1 Related work and discussion

The basic ideas for plan-merging and step-merging are in [Yang 1992; Yang, Nau et al. 1992; Yang 1997]. Yang implemented a solver that can handle partial-ordering constraints for temporal variables, as well as codesignation, and non-codesignation constraints for predicate variables. Yang

also defines a meta-CSP to handle the disjunct selection, checks the temporal consistency or the ordering constraints using a transitive closure structure, and similar structures for the consistency of the binding constraints. In addition he defines the relation *inconsistent*, *value subsumption,* and *variable subsumption*, which are very similar to the ones used by Epilitis in Chapter 4. He then uses these concepts to extend arc-consistency, used in preprocessing and during search. Actually, Epilitis started as an extension of Yang's solver. Even though Yang's solver is not mentioned in the rest of the DTP literature, it is an ancestor and precursor of DTP solving and it shares many similarities with the DTP solvers.

Our approach to the treatment of plan-merging, the definition of conflicts, and the addition of conflict-resolution constraints is based again on Yang's work. We extended Yang's work however in several dimensions:

- Our constraint solver, Epilitis, can handle quantitative temporal constraints and employs a number of pruning methods not found in Yang's solver. On the other hand the implemented version of Epilitis cannot at the moment handle binding constraints (but in Section 7.4.1 we discussed how to modify it to this extent).
- We have extended plan-merging algorithms to handle plans with quantitative temporal constraints and plans with conditional branches.
- We have extended plan-merging algorithms to consider plan-merging and step-merging as one step. In contrast, Yang separates the two steps: first he performs plan-merging on two plans ignoring step-merging as a conflict resolution method and as a way of reducing the overall cost of the plans. He subsequently performs step-merging.

These extensions satisfy a number of desiderata and predictions as expressed by Yang: "We have instead taken the conceptually clearer approach of separately discussing plan merging [i.e. step-merging using our terminology] and conflict resolution [i.e. plan-merging]. It would be worthwhile to see how plan-merging and conflict resolution could be fruitfully combined in a seamlessly way" [Yang 1997], page 139, and "One advantage of our theory is its extensibility; with a more elaborate planning language [such as allowing DTP and CDTP types of constraints in the plans], the underlying theory for global conflict resolution [plan merging] need not change", [Yang 1997], page 118.

Yang's work on step-merging is summarized in an optimal and approximate algorithm for performing step-merging. As already mentioned, the step-merging phase is clearly separated from the plan-merging phase. Yang considers step merges in which a whole set of steps S_1 is substituted by another set of steps S_2 , i.e. he removes our restriction of only allowing merging a pair of steps. However, he imposes the restriction that the steps in S_1 and S_2 are totally ordered and no other step in the plan can come between them. His algorithms are based on dynamic programming.

The problem is cost-in-context as presented here appeared in [Horty and Pollack 1998; Horty and Pollack 2001]. Horty and Pollack's approach however does not handle plans with quantitative temporal constraints and it separates the phases of plan-merging and step-merging. The cost-in-context algorithm maintains and continuously updates upper and lower bounds for $Cost(P_2 \mid P_1)$ and stops search as soon as these bounds determine the outcome of the comparison $Utility(P_2) - Cost(P_2 \mid P_1) > 0$. For example, if the upper-bound *UB* for $Cost(P_2 \mid P_1)$ is such that $Utility(P_2) - UB > 0$ the plan is adopted immediately and the search for the optimal step-merges is stopped. Their algorithm calls as a subroutine a plan-merging algorithm that is actually the first version of Epilitis

used in the work reported at [Tsamardinos, Pollack et al. 2000]. The algorithm **Minimum-cost-Merge** presented here could employ the idea of maintaining bounds for the $Cost(P_2 \mid P_1)$ if Epilitis search is extended with a branch and bound technique. We intend to perform such an extension for Epilitis in the future and combine the ideas in this chapter and in Horty and Pollack's work.

7.6.2 Contributions

The contributions in this chapter can be summarized in the following:

- We showed how constraint solvers and consistency checking algorithms, such as the ones for DTP and CDTP presented in the previous chapters, can be used to solve plan-merging, step-merging, and cost-in-context evaluation problems in planning, in a seamless and conceptually clean way.
- We presented algorithms that solve these problems, namely plan-merging, step-merging, and cost-in-context evaluation of a plan.

We believe that the algorithms we presented demonstrate in an impressive way the conceptual power of constraint solving techniques to plan merging problems. All the algorithms we presented take but a few lines. By transforming the problems to constraint satisfaction problems we removed all the technical details from the presentation of the algorithms. Moreover, any improvement discovered for the constraint solvers directly applies to the merging algorithms we presented. Even though we have not shown experimentally that the transformation to CSPs also has computational benefits, it is a recurring theme in planning that by transforming planning problems to CSPs we get impressive performance results. For example see the work by [Kautz and Selman 1992; Joslin 1996; Blum and Furst 1997; Weld 1999; Kambhampati 2000].

7.6.3 Future work

There are numerous of applications and extensions of this work. Obviously, we would very much like experimental comparison of our methods with Yang's plan-merging and step-merging algorithms, and with Horty and Pollack's algorithm for cost-in-context. In addition, we would like to improve the efficiency of the algorithms we presented by extending them with a branch and bound search method for the cost-in-context problem.

8. CONCLUSIONS

8.1 Summary

The problem we addressed in this dissertation was to provide efficient temporal reasoning algorithms and expressive temporal reasoning formalisms to fill the needs of a number of applications. We showed their applicability by providing conceptually clear algorithms to a number of previously unsolved, but interesting and important, planning problems using these new techniques.

We employed constraint-based temporal reasoning formalisms in which reasoning with the temporal primitives (i.e. time-points or intervals or both) can be done in a purely algebraic model. We also borrowed and adopted a number of key ideas from the Constraint Satisfaction literature to boost the efficiency of our algorithms.

Our motivation and inspiration came from real planning problems and systems but the potential applications of this work spread over a wide range of systems. Previous versions of our algorithms have already been applied in two planning projects we have been working on, the Plan Management Agent, an intelligent calendar application that manages the user's plans, and Nursebot, a robotic assistant indented to help the elderly with their daily activities.

The work presented in this dissertation contributes to previous state-of-the-art formalisms by *providing more efficient consistency checking algorithms* and *a first dispatching algorithm,* fundamental reasoning tasks for temporal problems. We then extended the *expressivity* of these formalisms to be able to encode conditional execution branches and so capture the uncertainty of the outcome of observations. Along with the definitions, we provide the appropriate consistency checking algorithms. Subsequently, we turned our attention from purely temporal reasoning problems to a number of planning problems as an application area for our new techniques. We showed how several important planning problems could be converted to instances of the new classes of temporal problems we developed and so have conceptually clear solutions. The transformation also has potential computational advantages.

Specifically, we presented techniques that significantly improve the efficiency of consistency checking in DTPs, implementing these techniques in a DTP solver named *Epilitis*. We also presented a dispatching algorithm for DTPs. We improved the expressiveness of constraint-based temporal problems by defining two new classes of problems, Conditional Simple Temporal Problems (**CSTP**) and Conditional Disjunctive Temporal Problems (**CDTP**). These can be used to represent flexible, quantitative temporal constraints in conditional execution contexts. We then developed the appropriate consistency checking algorithms for CSTPs and CDTPs and identify efficiently solvable subclasses of these problems. Interestingly, the efficiently solvable subclasses align with traditional conditional planning problems.

As a further result, since temporal problems can represent plans, the increased expressivity of our formalisms has direct consequences for *planning*. We showed how several planning problems can be transformed into our CSTP and CDTP formalism. In particular, we show how the constraint-based temporal approach can be used for *plan merging, plan cost optimization, and plan cost evaluation in context* with richly expressive plans that include quantitative temporal constraints and conditional branches.

From an engineering perspective, we continue the incorporation of the results in our two on-going projects, the Plan Management Agent and Nursebot. We are also in the process of identifying potential applications of this work to other areas such as Scheduling, Workflow Management Systems, Virtual Enterprises, and Temporal Databases. We would like to extend the new formalisms to reach new levels of expressiveness by using decision theoretic techniques and features adopted from other formalisms.

8.2 Contributions

In this section we present in more detail our contributions to the area. On the problem of DTP consistency checking, we first presented a thorough survey of the area, all the previous DTP solvers, algorithms, and specific search pruning methods. We classified the solvers and identified the particular trade-offs they each favor. Then, we analyzed the interactions of all previous pruning methods so to be able to combine them in one algorithm that utilizes all of these ideas. We also experimented with each pruning method and their combinations to discover their relative power. We introduced to DTP-solving a new pruning technique, called no-good recording, adopted from Constraint Satisfaction. We showed empirically that it is one of the most powerful pruning methods. Not only do no-goods efficiently and very effectively prune the search space, but they can also be used as heuristic information. We defined and experimented with a set of heuristics that utilize this information to identify the best heuristic. All together, the results of the theoretical analysis on DTP solving, the no-good pruning method, the ability to combine and utilize all previous pruning methods, and the new heuristics resulted in a DTP solver that is about two orders-of-magnitude faster than the previous state-of-the-art solver on standard DTP benchmarks. The algorithm has been fully implemented and tested in a system called Epilitis.

Another important result on DTP solving is the theoretical comparison of the CSP versus SAT approach. In the former, the meta-problem of selecting which disjunct to satisfy from each disjunctive DTP constraint forms a CSP problem, while in the latter approach it forms a SAT problem. We also present experimental evidence that support the hypothesis that the number of consistency checks is the wrong machine-independent and implementation-independent measure of performance in DTP solving. The number of consistency checks was up until now the standard measure of performance.

On DTP dispatching, we first presented the problem and defined the desirable properties a dispatcher should have, i.e. being correct, deadlock-free, maximally flexible, and useful. We then proceeded to design an algorithm that implements such a dispatcher and prove it has these properties. Given this algorithm, it is now possible to execute plans that are encoded as DTPs, while maintaining all the temporal flexibility of the occurrence of events during execution.

We then proceeded to examine temporal reasoning with conditional events. We defined two new formalisms named the Conditional Simple Temporal Problem (CSTP) and the Conditional Disjunctive Temporal Problem (CDTP) that are extensions of the STP and DTP respectively. In addition to the features their ancestral formalisms can represent, CSTP and CDTP allow conditional events, i.e. events that only occur under certain conditions. Thus, these classes of problems can be used to represent conditional and temporal plans.

Three notions of consistency were identified in CSTP and CDTP, namely Strong, Weak, and Dynamic Consistency. Strong Consistency in CSTP was proved equivalent to STP consistency and thus it is polynomial. Equivalently, Strong Consistency in CDTP is equivalent to DTP consistency and thus it is NP-complete. Weak CSTP consistency was proved co-NP-complete and algorithms for Weak Consistency were provided. The most complex and interesting case of Dynamic Consistency was addressed and a consistency checking algorithm was designed. The algorithm is intractable in the general case, but structural assumptions that enable a more efficient Dynamic consistency checking were identified. The structural assumptions align with traditional conditional planning problems.

The next problem we addressed was the application of the temporal reasoning results to planning. The new formalisms enabled for the first time the representation of plans with flexible, quantitative temporal constraints and conditional execution contexts. We showed how a number of interesting and important planning problems could be converted to temporal problems in a conceptually clear way. Using CSTPs and CDTPs the transformation is applicable to richly expressive plans. Planning problems on simpler plans without conditional execution context are transformed to DTPs instead and so can potentially be solved very efficiently with our Epilitis solver. The particular problems we addressed are plan merging, where a number of plans are merged into one in a way that avoids negative interactions, plan cost optimization, where steps in a plan are merged in a way that maximally reduces the plan cost, and plan cost evaluation in context, where the cost of a plan executed in the context of another plan is calculated to decide whether the plan should be adopted or not.

The contributions of this dissertation to temporal reasoning and planning have their independent merit. DTP solving, for example, is an interesting temporal reasoning problem on its own, even without considering its applications to planning that we presented. However, in considering the contributions collectively, an idea emerges, that of applying constraint-based temporal reasoning problems first to represent the plans and then to convert the planning problems into a DTP problem.

8.3 Future Work

There is a plethora of directions to extend and improve on this work. Improving the performance of Epilitis is one of them. The DTP is a newly introduced formalism and thus there is plenty of room for new ideas. We only provided one no-good recording algorithm, but others are possible, such as those based on restarts. Better heuristics are also possible. We predict that preprocessing techniques will play an important role in DTP solving, especially in domains where there is special structure in the DTP instances. Scheduling techniques, such as profiling, could possibly find applications in DTP solving.

It is also interesting to investigate algorithms for solving simple extensions of the DTP. One of them is the "generalized" DTP in which each disjunct is a conjunction of STP constraints. The problem can be converted to a DTP but further research should determine what is the most efficient way to solve this type of problems. Similarly, CSTPs and CDTPs can be transformed to DTPs but special techniques should be developed to take advantage of the special structure in the DTPs that are the results of the conversions. Plan merging problems are also converted to DTPs and thus pose the same research problem.

Another extension of the DTP is Dynamic DTPs, i.e. sequences of similar DTPs that differ only by a few constraints. We conjecture that Dynamic DTPs are important because they will arise in a series of plan merging problems and temporal plan generation. The no-goods discovered during the search to solve the problems in the series can be used for solving the next DTP in the series, potentially much faster. A final DTP extension that we would like to mention is one in which the DTP is augmented with codesignation and non-codesignation constraints. This extension is required so that plan merging problems with plans that contain n-ary predicates can be converted to DTPs. DTPs are very expressive and thus a number of other problems, such as scheduling and TCSP problems, can be cast as DTPs. A comparison of solving these problems as DTPs versus specialized algorithms would be very interesting.

The algorithm we presented at Chapter 5 on DTP dispatching is the first of its kind. As such, it is expected that a number of improvements are possible. Our algorithm satisfies the desired properties under certain assumptions, the most restrictive one being that there is enough processing time for it to run under any circumstance. It will be interesting to solve the problem under real-time constraints. For example, how much flexibility should one keep (under some measure of flexibility) so that the calculations can be performed in time to satisfy the real-time constraints? Another open issue on the subject is to measure the performance of our algorithm and provide efficient implementations.

The work on temporal constraints and conditional events is fairly novel and as such opens up a new range of research problems. First of all, what are the areas that can benefit from this type of temporal problems, other than planning? Efficient CSTP and CDTP consistency checking is an interesting line of research to pursue. We presented one set of structural assumptions that lead to efficient consistency checking. However, identifying other tractable subclasses is very important since the general case is computationally very hard and different applications form structurally different CSTPs and CDTPs.

We are currently investigating extensions to DTP, CSTP, and CDTP that include features that can encode uncertainty as to the occurrence of events. Such uncertainty can be captured in the STPU formalism and thus combining the formalisms into one will define a very powerful and expressive class of problems. The STPU essentially models the uncertainty of uncontrollable events (i.e. an event whose exact timing is outside the control of the agent) with uniform distributions. A more general decision theoretic framework could perhaps be applied to give rise to other formalisms that deal with uncertainty and can deal with probabilistic constraints and utility function. A decision theoretic perspective can also be the source of many ideas for the CSTP and CDTP where the uncertainty regarding the outcome of observations (i.e. the truth value of the conditions) gets associated with probability distributions.

Focused on planning, we presented algorithms for plan merging, plan cost optimization, and plan cost evaluation in context. Plan merging is basically conflict resolution and in turn, conflict resolution is an operation that appears during plan generation. Thus, our work on plan merging is the first step to build a temporal and conditional planner. For a planner that searches in the space of plans, e.g. UCPOP, in every search node it either adds a new step, reuses an old step to satisfy an open precondition, or resolves a conflict. Such a planner could generate temporal plans if it restrains from conflict resolution during search and only when there are no open preconditions, resolve all conflicts by converting them to a DTP or CDTP and passing them to Epilitis or another appropriate solver.

Once such a temporal planner is built, one could model the planning itself as a temporally extended step and "plan to plan". This ideas was first employed in the Remote Agent, but it has not been researched thoroughly even though it forms an interesting general problem: how to plan while (self)reasoning about the time it takes for the planning operation itself.

Other future work we intend to perform is the application of our planning algorithms to Workflow Management Systems and the forming of Virtual Enterprises. Workflows are essentially plans and when new workflows are committed to in a Workflow Management System this is in essence a plan merging operation. Similarly, when a Virtual Enterprise is formed, we need to ensure that the resulting workflow, which is the merge of a number of plans, is consistent. Similar applications are found in the consistency checking of clinical protocols: a patient that is diagnoses with more than one disease is simultaneously on many different treatment protocols whose collective consistency needs to be checked. Other potential application areas are Scheduling and Temporal Databases.

It is an exciting time for this area of research. A number of real projects are underway that exemplify the use of planning with metric time. The work in this dissertation and its future extensions will hopefully be part of this effort.

BIBLIOGRAPHY

BIBLIOGRAPHY

1. Aliferis, C. F. and G. F. Cooper (1996). A Structurally and Temporally Extended Bayesian Belief Network Model: Definitions, Properties, and Modeling Techniques. Uncertainty in Artificial Intelligence (UAI'96).

2. Allen, J. (1983). "Maintaining knowledge about temporal intervals." Communications of the ACM **26**: 832-843.

3. Allen, J. (1991). Planning as Temporal Reasoning. KR'91.

4. Allen, J. and J. Koomen (1990). Planning using a temporal world model. Readings in Planning. J. Allen, J. Hendler and A. Tate, Morgan Kaufmann Publishers**:** 559-565.

5. Armando, A., C. Castellini, et al. (1999). SAT-based Procedures for Temporal Reasoning. 5th European Conference on Planning (ECP-99).

6. Bacchus, F. and P. v. Beek (1998). On the conversion between non-binary and binary constraint satisfaction problems. National Conference on Artificial Intelligence (AAAI'98), Madison, WI.

7. Bacchus, F. and F. Kabanza (1996). Planning for Temporally Extended Goals. Proceedings of the 13th National Conference on Artificial Intelligence.

8. Bacchus, F. and F. Kabanza (1996). Using temporal logic to control search in a forward chaining planner. New Directions in Planning. M. Ghallab and A. Milani. Amsterdam, IOS Press**:** 141-153.

9. Barber, F. (2000). "Reasoning on Interval and Point-based Disjunctive Metric Constraints in Temporal Contexts." Journal of Artificial Intelligence Research **12**: 35-86.

10. Beek, P. V. (1989). Approximation algorithms for temporal reasoning. International Joint Conference in Artificial Intelligence (IJCAI'89).

11. Bessiere, C. (1999). Non-binary constraints. Principles and Practice of Constraint Programming (CP'99), Alexandria, Virginia, USA, Springer.

12. Bessiere, C., P. Mesequer, et al. (1999). On forward-checking for non-binary constraint satisfaction. CP'99, Alexandria, VA.

13. Bessiere, C. and J. C. Regin (1997). Arc consistency for general constraint networks: preliminary results. International Joint Conference on Artificial Ingelligence (IJCAI'97(), Nagoya, Japan.

14. Blum, A. L. and M. L. Furst (1997). "Fast Planning through Planning Graph Analysis." Artificial Intelligence **90**: 281-300.

15. Boutilier, C., T. Dean, et al. (1995). Planning under uncertainty: structural assumptions and computational leverage. Proceedings of the European Workshop on Planning.

16. Boutilier, C., T. Dean, et al. (1999). "Decision Theoretic Planning: Structural Assumptions and Computational Leverage." Journal of Artificial Intelligence Research.

17. Brusoni, V., L. Console, et al. (1997). An efficient algorithm for temporal abduction. AI*IA**:** 195-206.

18. Brusoni, V., L. Console, et al. (1997). "Later: Managing Temporal Information Efficiently." IEEE Expert **12**(4): 56-63.

19. Burch, J. R., E. M. Clarke, et al. (1994). "Symbolic Model Checking for Sequantial Circuit Verification." IEEE Transactions on CAD **13**: 401-424.

20. Cesta, A. and A. Oddi (1996). Gaining Efficiency and Flexibility in the Simple Temporal Problem. Third International Workshop on Temporal Representation and Reasoning (TIME-96), Key West, FL, USA, IEEE Computer Society Press,Los Alamitos, CA, USA.

21. Chen, X. and P. v. Beek (2001). "Conflict-Directed Backjumping Revisited." Journal of Artificial Intelligence Research **14**: 53-81.

22. Cheng, C. and S. F. Smith (1995). Applying constraint satisfaction techniques to Job-Shop Scheduling. Pittsburgh, Carnegie Mellon University.

23. Cheng, C. and S. F. Smith (1995). A constraint posting framework for scheduling under complex constraints. Joint IEEE/INRIA conference on Emerging Technologies for Factory Automation, Paris, France.

24. Chleq, N. (1995). Efficient Algorithms for Networks of Quantitative Temporal Constraints. Constraints'95.

25. Cimatti, A. and M. Roveri (1999). Conformant Planning via model checking. European Conference in Planning (ECP'99).

26. Clarke, E. M. and E. A. Emerson (1981). Design and Synthesis of synchronization skeletons using Branching Time Temporal Logic. Workshop on Logics of Programs, LNCS.

27. Coradeschi, S. and T. Vidal (1998). Accounting for Temporal Evolutions in Highly Reactive Decision-Making. TIME-98.

28. Cormen, T. H., C. E. Leiserson, et al. (1990). Introduction to Algorithms. Cambridge, MA, MIT Press.

29. Dean, T. L. and D. McDermott (1987). "Temporal Data Base Management." Artificial Intelligence **32**: 1-55.

30. Dechter, R. (1990). "Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition." Artificial Intellience **41**.

31. Dechter, R. and A. Dechter (1998). Belief Maintainance in dynamic constraint networks. National Conference in Artificial Intelligence (AAAI-98), St. Paul, MN.

32. Dechter, R. and D. Frost (1999). Backtracking algorithms for constraint satisfaction problems, University of California at Irvine.

33. Dechter, R., I. Meiri, et al. (1991). "Temporal constraint networks." Artificial Intelligence **49**: 61-95.

34. Dousson, C., P. Gaborit, et al. (1993). Situation Recognition: Representation and Algorithms. International Joint Conference on Artificial Intelligence (IJCAI'93).

35. Draper, D., S. Hanks, et al. (1994). Probabilistic planning with information gathering and Contingent Execution. 2nd International Conference on Artificial Intelligence Planning Systems (AIPS'94).

36. Du, D., J. Gu, et al. (1997). Satisfiability problem : theory and applications : DIMACS workshop, March 11-13, 1996. Providence, R.I., American Mathematical Society.

37. Durfee, E. H. and V. Lesser (1987). Using partial global plans to coordinate distributed problem solvers. International Joint Conference on Artificial Intelligence (IJCAI-87), Morgan Kaufmann.

38. Erol, K., J. Hendler, et al. (1994). HTN Planning: Complexity and Expressivity. Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94). Seattle.

39. Etzioni, O., S. Hanks, et al. (1992). An approach to planning with incomplete information. Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning**:** 115-125.

40. Ferguson, G. (1991). Domain Plan Reasoning in TRAINS-90, Department of Computer Science, University of Rochester.

41. Ferguson, G., J. Allen, et al. (1996). TRAINS-95: Towards a mixed initiative planning assistant. Artificial Intelligence Planning Systems (AIPS'96).

42. Fox, M. (1987). Constraint-directed search: a case study of Job-Shop Scheduling. San Mateo, CA, Morgan Kaufmann.

43. Frost, D. (October 1997). Algorithms and Heuristics for Constraint Satisfaction Problem. ICS, UCI.

44. Frost, D. and R. Dechter (1994). Dead-end driven learning. National Conference in Artificial Intelligence (AAAI-94).

45. Gaborit, A. P. A. and C. Sayettat (1990). Semantics and validation procedures of a multi-modal logic for formalization of multi-agent universes. European Conference in Artificial Intelligence (ECAI'90).

46. Galton, A. (1987). Temporal Logics and their Applications, Academic Press.

47. Gennari, R. (1998). Temporal Reasoning and Constraint Programming: A Survey. Institute for Logic Language and Computation. Amsterdam, University of Amsterdam.

48. Gerevini, A. and M. Cristani (1997). On finding a solution in temporal constraint satisfaction problems. International Joint Conference on ARtificial Intelligence (IJCAI'97), Nagoya, Japan.

49. Gerevini, A. and L. Schubert (1995). "Efficient algorithms for qualitative reasoning about time." Artificial Intelligence **74**: 207-248.

50. Ghallab, M. and H. e. Laruelle (1994). Representation and Control in IxTeT, a Temporal Planner. Proceedings of the Second International Conference on Artificial Intelligence Planning Systems (AIPS-94)**:** 61-67.

51. Ginsberg, M. (1993). "Dynamic Backtracking." Journal of Artificial Intelligence Research.

52. Ginsberg, M. and D. McAllester (1994). "GSAT and Dynamic Backtracking." Journal of Artificial Intelligence Research.

53. Golden, K. (1998). Leap before you look: information gathering in the PUCCINI planner. Artificial Intelligence Planning Systems (AIPS'1998).

54. Golden, K. and D. Weld (1996). Representing sensing actions: the middle ground revisited. Principles of Knowledge Representation and Reasoning (KR'96).

55. Gomez, C. P., B. Selman, et al. (1998). Boosting combinatorial search through randomization. National Conference on Artificial Intelligence (AAAI'98), Madison, Winsonsin.

56. Grasso, E., L. Lesmo, et al. (1990). Semantic interpretation of tense, actionality, and aspect. European Conference in Artificial Intelligence (ECAI'90).

57. Haddawy, P. and S. Hanks (1998). "Utility Models for Goal-Directed Decision-Theoretic Planners." Computational Intelligence **14**(3): 392-429.

58. Halpern, J. Y. and Y. Shoham (1986). A propositional modal logic of time intervals. Symposium on Logic in Computer Science.

59. Hanks, S. and D. S. Weld (1992). Systematic Adaptation for Case-Based Planning. Proceedings of the First International Confernce on AI Planning Systems, Morgan Kaufmann**:** 96-105.

60. Haralick, R. and G. Elliot (1980). "Increasing tree search efficiency for constraint satisfaction problems." Artificial Intelligence.

61. Haugh, B. A. (1987). Non-standard semanticns for the method of temporal arguments. Interantional Joint Conference in Artificial Intelligence (IJCAI'87).

62. Hayes, C. C. (1989). A model of planning for plan efficiency: taking advantage of operator overlap. National Conference in Artificial Intelligence, Detroit, MI, Morgan Kaufmann.

63. Hentenrick, P. v. (1989). Constraint Satisfaction in Logic Programming, MIT Press.

64. Horty, J. F. and M. E. Pollack (1998). Option Evaluation in Context. Proceedings of the 7th Conference on Theoretical Aspects of Rationality and Knowledge (TARK-98). San Francisco, Morgan Kaufmann**:** 249-262.

65. Horty, J. F. and M. E. Pollack (2001). "Evaluating New Options in the Context of Existing Plans." Artificial Intelligence **127**(2): 199-220.

66. Jaffar, J. and J. Lassez (1994). "Constraint Logic Programming : A survey." Journal of Logic Programming **19**(20): 503-581.

67. Jensen, R. M. and M. M. Veloso (2000). "OBDD-based Universal Planning for Synchronized Agents in Non-Deterministic Domains." Journal of Artificial Intelligence Research **13**: 189-226.

68. Jonsson, A., P. Morris, et al. (1999). Next generation Remote Agent planner. International Symposiumon AI, Robotics, and Automation in Space, Noordwijk, Netherlands.

69. Jonsson, A., P. Morris, et al. (2000). Planning in interplanetary space: theory and practice. Artificial Intelligence Planning and Scheduling (AIPS-00).

70. Jonsson, P., P. Haslum, et al. (2000). "Towards Efficient Universal Planning: A randomized approach." Artificial Intelligence **117**: 1-29.

71. Joslin, D. (1996). Passive and Active Decision Postponement in Plan Generation, Intelligent Systems Program, University of Pittsburgh.

72. Joslin, D. and M. E. Pollack (1995). Active and Passive Postponement of Decisions in Plan Generation. Proceedings of the Third European Workshop on Planning.

73. Kambhampati, S. (2000). "Planning Graph as (dynamic) CSP: Exploiting EBL, DDB, and other CSP techniques in Graphplan." Journal of Artificial Intelligence Research.

74. Kautz, H. and B. Selman (1992). Planning as Satisfiability. Proceedings of the European Conference on Artificial Intelligence. Vienna, Austria**:** 359-363.

75. Kautz, H. and B. Selman (1996). Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search. Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI). Portland, OR.

76. Kautz, H. and J. Walser (1999). "State-space planning by integer optimization." Knowledge Engineering Review **15**(1).

77. Keravnov, E. and J. Washbrook (1990). "A temporal reasoning framework used in the diagnosis of skeletal dysplasias." Artificial Intelligence in Medicine **2**(5): 238-265.

78. Koehler, J. (1998). Planning under Resource Constraints. European Conference in Artificial Intelligence (ECAI'98).

79. Koubarakis, M. (1992). Dense time and temporal constraints with inequalities. KR-92.

80. Kripke, S. (1963). "Semantical considerations on modal logic." Acta Philosophica Fennica **16**: 83-94.

81. Kushmerick, N., S. Hanks, et al. (1995). "An Algorithm for Probabilistic Planning." Artificial Intelligence **76**: 239-286.

82. Laborie, P. and M. Ghallab (1995). Planning with sharable resource constraints. International Joint Conference in Artificial Intelligence (IJCAI'95).

83. Lansky, A. L. (1988). "Localized planning event-baesd reasoning for multiagent domains." Computational Intelligence **4**(4): 319-334-.

84. Mackworth, A. K. (1977). "Consistency in networks of relations." Artificial Intelligence **8**: 118-121.

85. Mantyla, M. and J. Opas (1988). Hutcapp - a machining operations planner. International Symposium on Robotics and Manufacturing systems.

86. McCarthy, J. and P. Hayes (1969). "Some philosophical problems from the standpoint of AI." Machine Intelligence **4**.

87. Meiri, I. (1990). Faster constraint satisfaction algorithms for temporal reasoning. Los Angeles, University of California, Los Angeles, Cognitive Systems Lab.

88. Meiri, I. (1991). Combining qualitative and quantitative constraints in temporal reasoning. National Conference in Artificial Intelligence (AAAI'91).

89. Meiri, I. (1992). Temporal Reasoning: A Constraint-Based Approach, UCLA.

90. Meiri, I. (1996). "Combining qualitative and quantitative constraint in temporal reasoning." Artificial Intelligence **87**(1-2): 343-385.

91. Minton, S., M. D. Johnston, et al. (1990). Solving Large-Scale Constraint Satisfaction and Scheduling Problems Using a Heuristic Repair Method. Proceedings of the Ninth National Conference on Artificial Intelligence. Boston, MA**:** 17-24.

92. Mittal, S. and B. Falkenhainer (1990). Dynamic Constraint Satisfaction Problems. National Conference on Artificial Intelligence (AAAI-1990).

93. Mohr, R. and T. C. Henderson (1986). "Arc-consistency and path-consistency revisited." Artificial Intelligence **28**(225-233).

94. Montanari, U. (1974). "Networks of Constraints: fundamental properties and applications to picture processing." Information Science **7**: 95-132.

95. Morris, P. and N. Muscettola (1999). Managing Temporal Uncertainty through Waypoint Controllability. Sixteenth International Joint Conference on Artificial Intelligence (IJCAI).

96. Morris, P. and N. Muscettola (2000). Execution of Temporal Plans with Uncertainty. 17th National Conference on Artificial Intelligence (AAAI-00).

97. Muscettola, N. (1994). HSTS: Integrating Planning and Scheduling. Intelligent Scheduling. M. Zweben and M. S. Fox. San Francisco, Morgan Kaufman**:** 169-212.

98. Muscettola, N., P. Morris, et al. (1998). Reformulating Temporal Plans for Efficient Execution. 6th Conference on Principles of Knowledge Representation and Reasoning (KR'98).

99. Muscettola, N., P. P. Nayak, et al. (1998). "Remote Agent: To Boldly Go where No AI System has Gone Before." Artificial Intellience **103**: 5-47.

100. Nadel, B. (1989). "Constraint Satisfaction Algorithms." Computational Intelligence.

101. Nebel, B. and H.-J. Burckert (1995). "Reasoning about temporal relations: a maximal tractable subclass of Allen's interval algebra." Journal of the ACM **42**(1): 43-66.

102. Oddi, A. and A. Cesta (2000). Incremental Forward Checking for the Disjunctive Temporal Problem. European Conference on Artificial Intelligence.

103. Onder, N. (1999). Contingency Selection in Plan Generation, University of Pittsburgh Department of Computer Science.

104. Onder, N. and M. E. Pollack (1999). Conditional, Probabilistic Planning: A Unifying Algorithm and Effective Search Control Mechanisms. Proceedings of the Sixteenth National Conference on Artificial Intelligence**:** 577-584.

105. Onder, N., M. E. Pollack, et al. (1998). A Unifying Algorithm for Conditional, Probabilistic Planning.

106. Orgun, M. A. and W. Ma (1994). An overview of temporal and modal logic programming. 1st International Conference on Temporal Logic, Springer-Verlag.

107. Pednault, E. P. D. (1989). ADL: Exploring the middle ground between STRIPS and the situation calculus. International Conference on Principles of Knowledge Representation and Reasoning.

108. Penberthy, J. S. and D. Weld (1992). UCPOP: A Sound, Complete, Partial-Order Planner for ADL. KR'92, Cambridge, MA.

109. Penberthy, J. S. and D. Weld (1994). Temporal Planning with Continuous Change. Proc. AAAI-94**:** 1010-1015.

110. Penczek, W. (1991). Branching Time and Partial Order in Temporal Logics. Warsaw, Poland, Polish Academy of Sciences.

111. Peot, M. and D. E. Smith (1992). Conditional Nonlinear Planning. Proceedings of the First International Conference on AI Planning Systems (AIPS-92). College Park, MD**:** 189-197.

112. Perkins, W. and A. Austing (1990). "Adding temporal reasoning to expert system building environments." IEEE Expert: 23-30.

113. Pnueli, A. (1981). "The temporal semantics of concurrent programs." Theoretical Computer Science **13**: 45-60.

114. Prior, A. (1955). "Diodoram modalities." Philosophical Quaterly **5**: 205-213.

115. Prosser, P. (1993). Forward Checking with Backmarking, University of Strathclyde.

116. Prosser, P. (1993). "Hybrid algorithms for the constraint satisfaction problem." Computational Intelligence **9**.

117. Pryor, L. and G. Collins (1993). Cassandra: Planning for contingencies, Institute for the Learning Sciences, Northwestern University.

118. Pryor, L. and G. Collins (1996). "Planning for Contingencies: A Decision-Based Approach." Journal of Artificial Intelligence Research **4**: 287-339.

119. Raghu Karinthi, D. S. Nau, et al. (1992). "Handling feature interactions in process-planning." Applied Artificial Intelligence **6**(4): 389-415.

120. Reichgelt, H. and N. Shadbolt (1990). A specification tool for planning systems. European Conference in Artificial Intelligence (ECAI'90).

121. Richards, E. T. (1998). Non-systematic Search and No-good Learning. IC Parc. London, Imperial College.

122. Rosenblitt, D. (1991). Supporting Collaborative Planning: The plan integration problem. Cambridge, MA, MIT.

123. Rucker, D. W., D. J. Maron, et al. (1990). "Temporal representation of clinical algorithms using expert systems." Computers and Biomedical Research **23**: 222-239.

124. Russ, T. A. (1990). "Using hindsight in medical decision making." Computer Methods and Programs in Biomedicine **32**: 81-90.

125. Russell, B. (1903). Principles of mathematics. London, George & Unwin.

126. Schiex, T. and G. Verfaillie (1994). "Nogood Recording for Static and Dynamic Constraint Satisfaction Problems." International Journal of Artificial Intelligence Tools **3**(2): 187 - 200.

127. Schiex, T. and G. Verfaillie (1994). Stubbornness: a possible enhancement for backjumping and no-good recording. European Conference in Artificial Intelligence (ECAI-94).

128. Schoppers, M. J. (1987). Universal Plans for Reactive Robots in Unpredictable Environments. Proceedings of the Tenth International Joint Conference on Artificial Intelligence**:** 1039-1046.

129. Schoppers, M. J. (1987). Universal Plans for Reactive Robots in Unpredictable Environments. 10th International Joint Conference on Artificial Intelligence.

130. Schwalb, E. and R. Dechter (1998). Temporal Reasoning with Constraints. ICS, University of California at Irvine.

131. Schwalb, E. M. (1998). Temporal Reasoning with Constraints. Information and Computer Science. Irvine, University of California at Irvine.

132. Sellis, T. (1988). "Multiple query optimization." ACM Transactions on Database Systems **13**(1).

133. Silva, J. P. M. and K. A. Sakallah (1996). GRASP - A new search algorithm for satisfiability. Ann Arbor, Michigan, University of Michigan.

134. Smith, B. M. (1995). A Tutorial on Constraint Programming, Division of Artificial Intelligence, School of Computer Studies, University of Leeds.

135. Smith, D. and D. Weld (1998). Conformant Graphplan. National Conference in Artificial Intelligence (AAAI'98).

136. Smith, D. E., J. Frank, et al. (2000). "Bridging the gap between planning and scheduling." Knowledge Engineering Review **15**(1).

137. Smith, D. E. and D. Weld (1999). Temporal planning with mutual exclusion reasoning. International Joint Conference on Artificial Intelligence (IJCAI'99).

138. Somenzi, F. (1999). "Binary Decision Diagrams." Calculational System Design, NATO Science Series F:Computer and Systems Sciences **173**: 303-336.

139. Srivastava, B. and S. Kambhampati (1999). Efficient planning through separate resource scheduling. AAAI Spring Symposium on search strategy under uncertainty and incomplete information, AAAI Press.

140. Staab, S. (1998). On Non-Binary Temporal Relations. European Conference on Artificial Intelligence.

141. Stefik, M. (1981). "Planning with Constraints." Artificial Intelligence **16**: 111-140.

142. Stergiou, K. and M. Koubarakis (1998). Backtracking Algorithms for Disjunctions of Temporal Constraints. 15th National Conference on Artificial Intelligence (AAAI-98).

143. Stergiou, K. and M. Koubarakis (2000). "Backtracking algorithms for disjunctions of temporal constaints." Artificial Intelligence **120**(1): 81-117.

144. Stergiou, K. and T. Walsh (1999). Encodings of non-binary constraint satisfaction problems. National Conference on Artificial Intelligence (AAAI'99), Orlando, FL.

145. Tsamardinos, I. (1998). Reformulating Temporal Plans for Efficient Execution. Pittsburgh, University of Pittsburgh.

146. Tsamardinos, I., P. Morris, et al. (1998). Fast Transformation of Temporal Plans for Efficient Execution. Proceedings of the 15th National Conference on Artificial Intelligence.

147. Tsamardinos, I., M. E. Pollack, et al. (1999). Adjustable Autonomy for a Plan Management Agent. AAAI Spring Symposium on Adjustable Agents.

148. Tsamardinos, I., M. E. Pollack, et al. (2000). Merging Plans with Quantitative Temporal Constraints, Temporally Extended Actions, and Conditional Branches. Artificial Intelligence Planning and Scheduling (AIPS'00), Breckenridge, Colorado, USA.

149. Tsang, E. (1987). TLP - a temporal planner. Advances in Artificial Intelligence, John Wiley & sons**:** 63-78.

150. Tsang, E. (1993). Foundations of Constraint Satisfaction, Academic Press.

151. Vere, S. A. (1983). "Planning in Time: Windows and Durations for Activities and Goals." IEEE Transactions on Pattern Analysis and Machine Intelligence **5**(3): 246-267.

152. Vidal, T. (2000). Controllability characterization and checking in Contingent Temporal Constraint Networks. 7th International Conference on Principles of Knowledge Representation and Reasoning (KR2000), Breckenridge, Colorado, USA.

153. Vidal, T. (2000). A unified dynamic approach for dealing with Temporal Uncertainty and Conditional Planning. 5th International Conference on Artificial Intelligence Planning and Scheduling (AIPS2000), Breckenridge, Colorado, USA.

154. Vidal, T. and S. Coradeschi (August 1999). Highly Reactive Decision-Making: a Game with Time. 16th International Joint Conference on AI (IJCAI-99), Stockholm, Sweden.

155. Vidal, T. and H. Fragier (1999). "Handling consistency in temporal constraint networks: from consistency to controllabilities." Journal of Experimental and Theoretical Artificial Intelligence **11**: 23-45.

156. Vidal, T. and M. Ghallab (1996). Dealing with Uncertain Durations in Temporal Constraint Networks Dedicated to Planning. The 12th European Conference on Artificial Intelligence**:** 48-52.

157. Vidal, T. and P. Morris (2001). Dynamic Control of Plans with Temporal Uncertainty. IJCAI-2001 (to appear).

158. Vila, L. (1994). "A survey on temporal reasoning in Artificial Intelligence." AI Communications **7**(1): 4-28.

159. Vilain, M. and H. Kautz (1986). Constraint propagation algorithms for temporal reasoning. National Conference in Artificial Intelligence (AAAI'86).

160. Wallace, R. and E. Freuder (2000). Dispatchable Execution of Schedules Involving Consumable Resources. Artificial Intelligence Planning and Scheduling 2000 (AIPS 200), Breckenridge, CO.

161. Weld, D., C. Anderson, et al. (1998). Extending graphplan to handle uncertainty and sensing actions. National Conference on Artificial Intelligence (AAAI'98).

162. Weld, D. S. (1994). "An Introduction to Least Commitment Planning." AI Magazine **15**(4): 27-61.

163. Weld, D. S. (1999). "Recent Advances in AI Planning." AI Magazine.

164. Wilkins, D. E. (1988). Practical Planning: Extending the Classical AI Paradigm. San Mateo, CA, Morgan Kaufmann.

165. Williamson, M. and S. Hanks (1994). Optimal Planning with a Goal-Directed Utility Model. Proceedings of the Second International Conference on Artificial Intelligence Planning Systems**:** 176-181.

166. Yang, B., R. E. Bryant, et al. (1998). A performance study of BDD-based model checking. Formal Methods on Computer-Aided Design.

167. Yang, Q. (1990). "Formalizing Planning Knowledge for a Hierarchical Planner." Computational Intelligence **6**: 12-24.

168. Yang, Q. (1992). "A Theory of Conflict Resolution in Planning." Artificial Intelligence **58**(1-3): 361-392.

169. Yang, Q. (1997). Intelligent Planning: A Decomposition and Abstraction Based Approach. New York, Springer.

170. Yang, Q., D. S. Nau, et al. (1992). "Merging Separately Generated Plans with Restricted Interactions." Computational Intelligence **8**(2): 648-676.

171. Yip, K. M. (1995). Tense, Aspect, and the Cognitive Representation of Time. International Joint Conference in Artificial Intelligence (IJCAI'95).

172. Yokoo, M. (1994). Weak-commitment search for solving constraint satisfaction problems. National Conference in Artificial Intelligence (AAAI-94).