Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2018-012 January 29, 2018

continuous Relaxation to Over-constrained Temporal Plans

Peng Yu

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Continuous Relaxation to Over-constrained Temporal Plans

by

Peng Yu

B.Eng., Hong Kong University of Science and Technology (2010)

Submitted to the Department of Aeronautics and Astronautics
in partial fulfillment of the requirements for the degree of

Master of Science in Aeronautics and Astronautics

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2013

Author .....
Department of Aeronautics and Astronautics
January 25, 2013

Certified by .....
Brian C. Williams
Professor
Thesis Supervisor

Accepted by .....
Eytan H. Modiano
Chairman, Department Committee on Graduate Theses

Continuous Relaxation to Over-constrained Temporal Plans

by

Peng Yu

Submitted to the Department of Aeronautics and Astronautics
on January 25, 2013, in partial fulfillment of the
requirements for the degree of
Master of Science in Aeronautics and Astronautics

Abstract

When humans fail to understand the capabilities of an autonomous system or its environmental limitations, they can jeopardize their objectives and the system by asking for unrealistic goals. The objective of this thesis is to enable consensus between human and autonomous system, by giving autonomous systems the ability to communicate to the user the reasons for goal failure and the relaxations to goals that archive feasibility. We represent our problem in the context of temporal plans, a set of timed activities that can represent the goals and constraints proposed by users. Over-constrained temporal plans are commonly encountered while operating autonomous and decision support systems, when user objectives are in conflict with the environment. Over constrained plans are addressed by relaxing goals and or constraints, such as delaying the arrival time of a trip, with some candidate relaxations being preferable to others. In this thesis we present Uhura, a temporal plan diagnosis and relaxation algorithm that is designed to take over-constrained input plans with temporal flexibility and contingencies, and generate temporal relaxations that make the input plan executable. We introduce two innovative approaches within Uhura: collaborative plan diagnosis and continuous relaxation. Uhura focuses on novel ways of satisfying three goals to make the plan relaxation process more convenient for the users: small perturbation, quick response and simple interaction.

First, to achieve small perturbation, Uhura resolves over-constrained temporal plans through partial relaxation of goals, more specifically, through the relaxation of schedules. Prior work on temporal relaxations takes an all-or-nothing approach in which timing constraints on goals, such as arrival times to destinations, are completely relaxed in the relaxations. The Continuous Temporal Relaxation method used by Uhura adjusts the temporal bounds of temporal constraints to minimizes the perturbation caused by the relaxations to the goals in the original plan.

Second, to achieve quick responses, Uhura introduces Best-first Conflict-directed Relaxation, a new method that efficiently enumerates alternative options in best-first order. The search space of alternative options to temporal planning problems is very large and finding the best one is a NP-hard problem. Uhura empirically demonstrates fast enumeration by unifying methods from minimal relaxation and conflict-directed

enumeration methods, first developed for model based diagnosis. Uhura achieves two orders of magnitude improvement in run-time performance relative to state-of-the-art approaches, making it applicable to a larger group of real-world scenarios with complex temporal plans.

Finally, to achieve simple interactions, Uhura presents to the user a small set of preferred relaxations in best-first order based on user preference models. By using minimal relaxations to represent alternative options, Uhura simplifies the options presented to the user and reduces the size of its results and improves their expressiveness. Previous work either generates minimal relaxations or full relaxations based on preference, but not minimal relaxations based on preference. Preferred minimal relaxations simplify the interaction in that the users do not have to consider any irrelevant information, and may reach an agreement with the autonomous system faster. Therefore it makes communication between robots and users more convenient and precise.

We have incorporated Uhura within an autonomous executive that collaborates with human operators to resolve over-constrained temporal plans. Its effectiveness has been demonstrated both in simulation and in hardware on a Personal Transportation System concept. The average runtime of Uhura on large problems with 200 activities is two order of magnitude lower compared to current approaches. In addition, Uhura has also been used in a driving assistant system to resolve conflicts in driving plans. We believe that Uhura's collaborative temporal plan diagnosis capability can benefit a wide range of applications, both within industrial applications and in our daily lives.

Thesis Supervisor: Brian C. Williams
Title: Professor

Acknowledgments

I would like to express my gratitude to all who have supported me in the completion of this thesis.

First of all, I would like to express my gratitude towards my supervisor, Professor Brian C. Williams, for his endless support and encouragement in the past two years. I came to MIT without any background in Model-based autonomy and artificial intelligence. He introduced me to the field, pointing in the right direction for my research and has provided invaluable feedback through numerous meetings and conversations, without with my thesis would not have been possible.

I would like to thank everyone in the MERS group for their insightful comments and discussions throughout my thesis writing, and for their support in my research in the past two years. Specifically, I thank David Wang, for his tireless help and answers to all my problems and questions, saving me from many frustrating situations. Eric, Steve, Andrew, Simon, Pedro, Shannon, Wesley, Larry and Bobby, thank you for helping me get up to speed in my research and get familiar with MIT. I am grateful to all of you for making my experience at MERS exciting and technically engaging. I would also like to thank all of my collaborators at Boeing Research & Technology for their guidance in the Personal Transportation System Project. A big thanks goes to Scott Smith and Ronald Provine for providing me many interesting ideas and valuable suggestions for my research directions.

Most importantly, I'd like to thank my parents Cui Jie and Yu Zhihe for their endless support and love throughout my life. Even though I am ten thousand kilometers away from home, you always encourage me to pursue my goals, whatever and wherever they may be. I also thank my girlfriend, Zhang Zhuo, for her love and guidance that helps me to overcome the difficulties in my life. Without their help, there is no way for me to come to MIT, pursuing my dream and finally completing this thesis.

Finally, I would like to thank my sponsor, the Boeing Company, for their generous support in the pass two years that makes this thesis possible. This project is supported

under Boeing Company grant MIT-BA-GTA-1. Additional support was provided by the DARPA meta program, under contract number 6923548.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As the performance of temporal planning algorithms improves over time, they have been incorporated into many planning and scheduling applications. However, a plan that can satisfy all of the user goals does not always exist. For example, a Mars rover may encounter an unexpected battery failure, leaving little time to complete its exploration task. Usually, planners will signal the user that a feasible plan that can satisfy all the goals cannot be found. However, it is not enough for the system to just signal a failure. When the complexity of the problem and plan increase, it becomes extremely difficult for humans to identify the resolutions. Therefore, the autonomous system or decision aid should explain the situation and propose alternative plans so that the engaged human operator can find a more informed resolution without too much effort. Specifically, the decision tool should offer key insights into the cause of failure and preferred plan repair options to the operator. For example, in the context of a Mars rover with a failed battery, we would expect the system to tell us which goals need to be dropped in order to guarantee a safe return to the base

This thesis develops Uhura, a temporal plan relaxation algorithm and system that addresses these issues. Uhura takes a mixed initiative approach that generates preferred minimal relaxations to over-subscribed temporal planning problems. It works with the human collaboratively towards the diagnoses of faulty plans. Uhura has three significant features compared to previous approaches: quick response, simple interaction and small perturbations. To support these features, we developed three

new methods in this thesis:

- First, Uhura minimizes the perturbation of relaxations to the original planning problem by continuously relaxing its goals specified by temporal constraints, which preserves all the plan elements in the output relaxed problem. For example, instead of about the mission completely, the rover informs the operator about an extended completion time.

- Second, Uhura resolves over-subscribed temporal planning problems through a conflict-directed diagnostic process, making it very efficient for relaxing large scale applications. A conflict can be viewed as a summary of cause of failure. In the Mars rover scenario, there is a conflict between the mission goals and limited battery power that makes the problem infeasible. To resolve a conflict, one must relax at least one goal in it, such as extending the mission completion time. A valid relaxation restores the feasibility of a planning problem by resolving all its conflicts.

- Third, Uhura only enumerates minimal relaxations, a compact representation of relaxations to over-subscribed planning problems. It reduces the size of results by orders of magnitude and significantly improves the run-time performance. For example, the rover will only asks the operator for either an extended mission time or a reduced set of goals, but not both.

We first provide an overview of the features and desired behaviors of Uhura through the trip planning problem of a Personal Transportation System, which is a form of robotic air taxi, in Section 1.1. Section 1.2 presents the current approaches to each claim and the technical challenges of their implementations. Finally, we describes the structure of the thesis in Section 1.3.

1.1 Motivation: Over-subscribed Problems are Everywhere

Nowadays, autonomous planning systems have been widely used in people's lives, especially in the fields of transportation and manufacturing. They have been used to generate the routes and schedules of flights, trains, buses and cars, and for generating work plans. Modern planning algorithms have demonstrated superior capabilities, especially for large scale problems that are beyond human decision making capabilities. For example, a planning and scheduling algorithm, *O-Plan*, has been used to generate production plans for Hitachi [6]. Their implementations have significantly reduced the workload of human operators and for optimizing operation efficiency .

A significant open challenge is to decide what to do when the situation is over-subscribed. For example, a Mars rover encounters an unexpected battery failure, leaving insufficient power for the rest of its mission. If a problem is over-subscribed, that is, no plan exists that can satisfy all the goals and requirements imposed by either human operators or the environment, these planners cannot help resolve such a problem. In this thesis, we introduce a novel approach to the over-subscribed problem based on the metaphor of *collaborative diagnosis*. Handling over subscription through collaborative diagnosis is based on two central claims:

- Handling over subscription is inherently a collaborative process. The operator knows the relative importance of different goals. It is unreasonable to expect that the operator will have presented this preference information to the planner a priori, and hence the planner will be able to decide the appropriate relaxation alone. Conversely, the human will need the planning tool to help explore the space of possible goal relaxations. The planner will have expertise and brute computational power that is better suited to this task.

- For the human to make informed decisions, the planner should be able to summarize the results of its reasoning processes to the human decision maker, as it pertains to the decisions that the human needs to make. This includes diag-

nostic information, such as why a set of goals cannot be feasibly achieved, and why a proposed relaxation addresses each of these identified concerns.

- We claim that over subscription can often be addressed with minimal disruption by relaxing constraints partially. We refer to this as continuous relaxation. The motivation is that the users usually want to minimize the perturbation to their goals and constraints made by the resolutions. Resolving over-subscribed problems by completely suspending user goals is unnecessary in most situations. A better way would be to adjust the user goals accordingly. For example, a student realizes that he cannot complete his problem set on time due to an approaching exam. Instead of giving up the exam or the problem set, he chooses to ask for an extension for his problem set, thus preserves his goals to the maximum.

The vision of this thesis is to provide an autonomous system that can detect the cause of failures in over-subscribed temporal plans, engage the human operators and provide suggestions for the repairs. The following three features are necessary for resolving over-subscribed problems: *quick response*, *simple interaction*, and *small perturbation*.

Quick response

The algorithm implemented in the diagnosis system should be efficient. Usually, people would expect an instant resolution coming out from the system if their plans are known to be broken, say within 1 to 2 seconds. Efficient algorithms help to implement quick response, and hence make the diagnosis process convenient for the users.

Simple interaction

The resolutions generated by the system must be compact and concise so that they can be communicated to the users easily. For example, in the Mars rover scenario, the operator would be more interested in the few goals that have to be dropped, not the ones that remain achievable. If multiple resolutions are

available, the system should be able to select the leading candidates preferred by the operator. Otherwise, it may take a long time for the operator to look through the long list of possible resolutions to a large scale problem. Moreover, the preference models should be easy to construct and evaluate.

Small perturbation

The resolutions generated by the system must minimize the perturbations made to the original problem. In other words, if an over-subscribed temporal planning problem can be resolved by removing one goal, the system should not suggest the user to remove more than that.

Collaborative diagnosis supports the first two features. It enables autonomous systems to provide quick response and simple user interaction. The extension to continuous relaxation enables user goals to be preserved to the maximum degree possible in the resolution to over-subscribed problems. We present a scenario in the following subsection to demonstrate the challenges and our approaches to the solution.

1.1.1 Planning a Trip Home Using the Personal Transportation System

Throughout this thesis, discussion will center around the example of the Personal Transportation System, a joint project between the Model-based Embedded and Robotic System group at MIT, the Boeing Company and the Center for the Study of Language and Information at Stanford University. This project aims at demonstrating the concept of an autonomous Personal Air Vehicle (PAV, Figure 1-1), and possibly on a vehicle similar in spirit to the Transition (Figure 1-2), in which the passenger interacts with the vehicle in the same manner that they interact with a taxi driver. To interact with a PAV, the passenger describes his/her goals and constraints in English. The autonomous system on-board the PAV checks the map and weather conditions, generates a safe plan and flies the vehicle to the destination. If there is a change in the weather condition or the destination airport is closed due

to flow control, the system can automatically adjust the original plan to achieve the passenger's goals.

The Temporal Plan Relaxation system, Uhura[1], is developed as he part of the project that supports collaborative diagnosis of over-constrained temporal plans. The state and temporal goals provided by the passenger may not be consistent for all times. Both the lack of knowledge about the vehicle and changes in environment constraints may cause inconsistency between the user's goals. However, the scale and complexity of the temporal plans are usually very large, making it difficult for a human being to detect the cause of failures and to find resolutions. Uhura is developed to work with the users collaboratively to resolve problems through schedule relaxations, similar to a travel agent. Uhura will propose alternatives to the user when an over-subscribed problem is identified, and negotiate with the user until an agreement is reached.

Figure 1-1: The Personal Air Vehicle simulated using X-Plane

Suppose that the passenger, John, just finished his work in the office and needs to be home in 60 minutes for a friend's birthday party. On his way back home, he would like to have dinner at a sandwich restaurant. The restaurant options he has are Cosi, Quiznos and Subway. Each restaurant has different waiting and dinner times, and the driving distances required by them are different as well. With the integration of a natural language dialog system developed by CSLI Stanford University [38], Uhura

[1]The name Uhura is from Star Trek.

Figure 1-2: The *Transition* flying car (Courtesy Terrafugia)

can interact with the user and capture all the goals through a simple conversation:

John:	This is John.
PAV:	Authorization Confirmed.
John:	I want to go home in 60 minutes.
PAV:	OK.
John:	I want to stop at a sandwich restaurant for dinner.
PAV:	OK. I found three sandwich restaurants on your way back: Cosi, Quiznos and Subway. Do you want to dine-in or take-out?
John:	Both are fine.
PAV:	Ok...
PAV:	I cannot find a plan for you.
PAV:	Your goal of arriving home in 60 minutes can not be achieved. There appear to be three good options, given your preferences. First, can you relax your goal to 85 mins?
John:	No.
PAV:	Then can you go home directly without having dinner?
John:	No.
PAV:	Can you accept a faster flying speed,

say 40 mph?

John: Ok, I think I can accept it this time.

PAV: OK, I have found a feasible plan for you.

John: Thank you PAV.

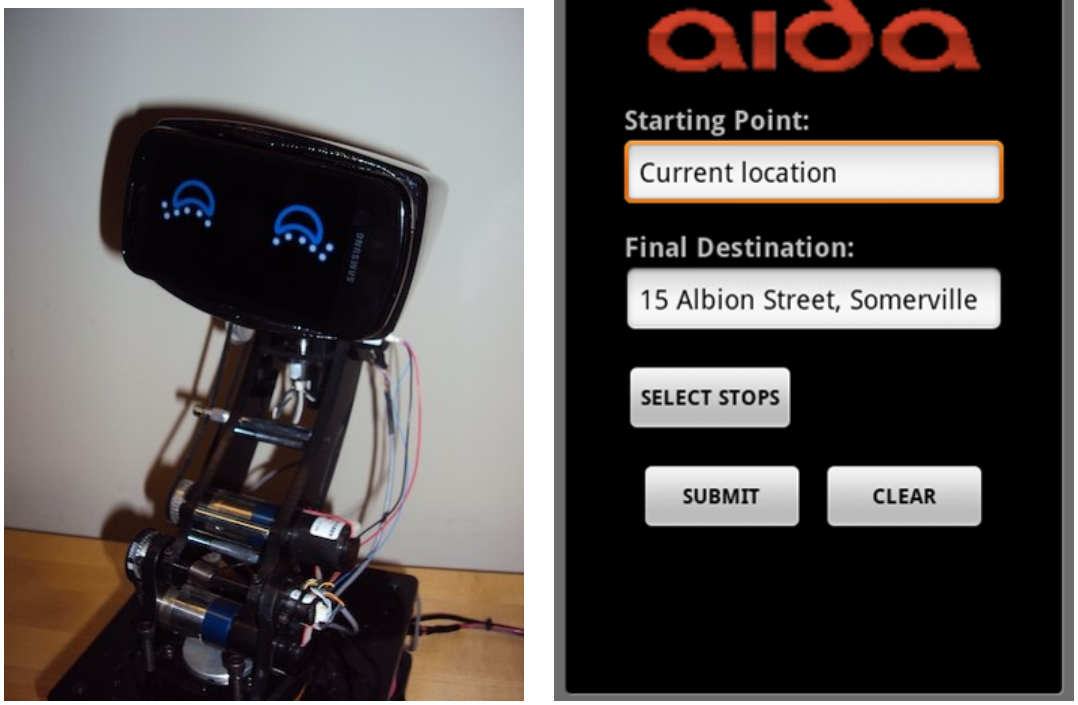
First, the planning algorithm of Uhura detects that no plan exists that can satisfy all of John's goals. Uhura then identifies the cause of failure: the temporal goal proposed by John (60 minutes) is too tight to complete all the tasks. Based on this conflict, collaborative diagnosis starts enumerating resolutions and engages John to negotiate for an alternative plan. For example, the PAV presents John an option that changes the trip duration. John rejects its proposal and Uhura continues the enumeration and looks for the next schedule relaxation that resolves the problem. Collaborative diagnosis continues proposing new resolutions until it reaches an agreement with John.

To make the collaborative process efficient, that is, reaching an agreement with the user as soon as possible, user preference models are used in the collaborative diagnosis algorithm. The end goal is for the user to select a relaxation that best meets the user's needs. Typically the space of feasible options is too large for the human to consider; instead the human would like to be presented with few good options. To do this the collaborative diagnostic algorithm needs to know the passengers' preferences. To address this requirement, Uhura generates a list of preferable repair options based on a metric cost function that encodes the passengers' preference over restaurant choices and the relaxations of schedule constraints. The user-preferred relaxations will be generated and presented first, hence shorten the negotiation process.

Second, the continuous relaxation algorithm post-processes the resolutions generated by the collaborative diagnosis algorithm and tries to preserve the user goals as much as possible. For example, the PAV notices that removing the duration constraint (60 minutes) can resolve the conflict in John's plan. Continuous relaxation then computes the minimal amount of adjustment to this constraint that is sufficient to resolve John's over-subscribed problem, without completely suspending this dura-

tion constraint. In this case, the constraint is relaxed from 60 minutes to 85 minutes, which is the resolution with the minimal perturbation.

In addition to the Personal Transportation System, Uhura has also been tested within many other applications, including a robotic driving assistant system, *AIDA (for Affective Intelligent Driving Agent)*, that provides suggestions to help drivers resolve timing conflicts in their trip plans (Figure 1-3).



(a) The AIDA Robot. (b) User interface of AIDA.

Figure 1-3: The AIDA robot (Courtesy MIT Media Lab [1]).

1.2 Related Work and Challenges

We outlined three goals for Uhura: quick response, simple interaction, and small perturbation, in Section 1.2. To address these goals, we introduce two innovative approaches to resolve over-subscribed temporal planning problems: Best-first Conflict-Directed Relaxation (BCDR) and continuous relaxation. In this section we present the technical challenges of implementing the approaches to satisfy these goals, and related work in the literature.

1.2.1 The Search Space is Enormous

Planning problems are generally very hard to solve due to the large numbers of possible states and activities, and their possible combinations. The same problem exists during the resolution of over-subscribed temporal planning problems. The number of possible resolutions is exponential: every state and temporal goal in the plan may be relaxed. For example, the temporal planning problem of booking a trip from Boston to Detroit has around 30 planning steps, and the number of possible resolutions can be as large as 10^{10}.

This enormous search space imposes a huge challenge to resolving over-subscribed temporal problems efficiently and to providing a quick response to the users. In fact, the problem of finding all the resolutions to an over-subscribed temporal planning problem is NP-Complete [29], assuming that a polynomial algorithm exists that can check if a temporal plan is executable.

Many techniques, especially the techniques developed to solve constraint satisfaction problems, have been implemented to speed up the search for resolutions, including standard and domain specific ones, such as forward checking, conflict-directed back jumping, Dualize & Advance [4], removal of subsumed variables [27] and semantic branching [3] (the last two techniques only apply to temporal problems). Temporal planning problems can be encoded using CSP formulations, hence enable the use of these techniques. One other approach is to give up the requirements on the completeness of the results and use a local search algorithm, like [5]. This approximate

approach usually runs much faster than the systematic methods, however, cannot guarantee the optimality or completeness of the results.

1.2.2 The Number of Resolutions is Far Beyond Human Reasoning Capability

The large numbers of results is also an issue for the users: facing thousands or even millions of resolutions, it is difficult for a human to select the correct one from them. This imposes a big challenge to resolving a problem through simple and efficient human machineinteraction. The autonomous system must be able to filter out unnecessary and less-preferred resolutions and only present a few preferred ones to the user, in order to let the user make an informed decision. In [29], an approach is presented to reduce the amount of resolutions generated by generating representative plan relaxations. It is based on the notion of *representative set*, in which all resolutions generated cannot be dominated by any other resolutions in the set.

In addition, most approaches choose to implement preference models to help resolve this issue. In [30], a real-valued cost function is associated with all the plan goals in order to evaluate and prioritize the resolutions generated by the algorithm. However, its preference function is restricted to discrete domain variables, in which constraints are either preserved or suspended. For continuously relaxed constraints, the preference function is more complex, since the relaxation has infinite numbers of states. For example, John's preference over the relaxation of duration constraint may depend on its extent. If the constraint is slightly relaxed, the relaxation is indifferent for John. On the other hand, if the constraint is relaxed by 100%, John may give up the whole trip due to his limited amount of time.

1.2.3 Perturbations to the User Goals Must Be Minimized

As stated before, the user would like to preserve his/her goals to the maximum, if possible, in the resolutions to an over-subscribed temporal planning problem. This challenge corresponds to the third requirement: small perturbation. As a valid reso-

lution, it must relax some of the user's goals on states and temporal constraints, like delaying the arrival time or removing way points from the trip. The perturbation to the user's goals is unavoidable.

Most of the previous work takes an all-or-nothing approach, in which user goals are suspended in order to resolve the over-subscription problem [16, 5, 31]. However, suspending goals can perturb a problem a significant amount, and is often unnecessary. For example, in John's trip, it would be unnecessary if the PAV asks him to remove his constraint on trip duration, since slightly relaxing the duration is enough to make his plan executable.

In [30], temporal relaxation is divided into several levels. For example, John's constraint on trip duration may be relaxed to 70, 80 or 90 minutes, depending on the over-subscription and John's preference. This approach preserves more plan elements than complete constraint relaxation, however, the quality of its resolutions highly depends on the discretization of the domain of users' goals.

1.3 Thesis Layout

This thesis presents two innovations that address the three challenges. First, we present the collaborative diagnosis algorithm, Best-first Conflict-directed Relaxation (BCDR), used in Uhura that enumerates minimal temporal relaxations and negotiates alternative options to over-subscribed temporal planning problems with the user. Through the implementations of conflict-directed best-first enumeration and minimal temporal relaxations, BCDR addresses the second and third requirements we presented in Section 1.1: simple interaction and small perturbation. It improves the efficiency of enumerating relaxation by two orders of magnitude compared to previous approaches. In addition, BCDR generates minimal relaxations, a compact representation of all relaxations to over-constrained temporal planning problems. The use of minimal relaxations significantly reduces the size of the search space and the results.

We present the algorithm in two steps: first we present a simpler version of BCDR that generates discrete relaxations. Then we present the continuous version of BCDR that maximizes the preservation of user goals. Unlike discrete relaxations to temporal constraints, continuous relaxations do not suspend any temporal constraint. Instead, it adjusts temporal constraints continuously until an executable plan can be generated. It can find the 'minimal' relaxation that is necessary for resolving over-subscribed problems, hence minimizes the perturbation to the users' goals.

In this thesis, we relax over-subscribed temporal planning problems. We achieve this by encoding them as inconsistent conditional temporal constraint networks, and by relaxing these constraints continuously. We demonstrate that the relaxation to the schedule of a planning problem is in fact equivalent to relaxing constraints in its equivalent temporal constraint problem, since each schedule constraint in the planning problem can be mapped to a unique temporal constraint in the constraint problem. BCDR is developed as a general constraint programming algorithm that can resolve any inconsistent conditional CSPs with discrete and continuous variables. It takes in an inconsistent problem and resolve all of its conflicts, by relaxing one or more of its

constraints.

In Chapter 2, we defines the related concepts used in this thesis, including the description of user goals (Qualitative State Plans), solutions (Temporal Planning Networks), cause of failure (Conflicts) and resolutions (Temporal Relaxations). In Chapter 3, we present the Best-first Conflict-directed Relaxation algorithm that enumerates minimal discrete relaxations to over-subscribed temporal problems. In Chapter 4, we describe the continuous version of BCDR and its integration with Uhura. In Chapter 5, we present the experimental results of BCDR on various benchmark problems. Finally, in Chapter 6 we summarize our work and discuss possible extensions to Uhura for future work.

Chapter 2

The Problem of Continuous Plan Relaxation

As discussed in Chapter 1, this thesis presents a general method for collaborative plan relaxation through diagnosis, and a more general, underlying method for continuously relaxing constraints on both discrete and real-valued variables. This method is demonstrated in the context of user interaction with a robotic air taxi. This chapter develops the problem statement, defines key supporting concepts and demonstrates each in the context of the Personal Transportation System scenario from Chapter 1.

As humans we are often inclined to do too much, and as a result discover that there is no way to achieve all of our goals. When this occurs, we consciously or unconsciously relax some of those goals until what remains is do able. Our problem is to provide an algorithm that aids a user in systematically exploring the space of goal relaxations. To turn this into a formal problem statement, we need to make precise the terms: goal, executable, relaxation and preference.

In our approach we view both goals and their executions as a form of temporal plan comprised of a set of activities to perform, such as go to the store, and constraints on their timing, such as depart in the next 30 minutes, and return within an hour. The difference between the plans used to describe goals and executions is their specificity. Goal plans provide general guidelines that are important to the user, such as have groceries in a hour, while an executable plan specifies concrete activities that we know

how to perform, such as turn in the car engine.

Given the central role of plans in goal relaxation, we begin by making precise these different concepts of plan and their execution. In a general planning problem, the goal is to generate a set of actions that can achieve all desired goals given a description of the environment, allowed actions and initial state. Usually, a planning problem involves three basic elements: A Planning Domain, A set of Goals and A Plan.

- A *Planning Domain* specifies a set of legal states and actions allowed in the planning problem.
- The *initial states* specify the status of the agents at time t=0.
- The *Goal* of a planning problem is a set of desired states at different times.
- A *Plan* is the solution to a planning problem, which involves a set of legal actions that, starting in the initial state, generate a set of states at different times that entails the *Goal*.

In this thesis, we use *Qualitative State Plans (QSP)* [24] to describe time evolved goal states given by the user. A QSP uses episodes and temporal constraints to specify the user goals, where episodes are constraints on state trajectories, over a bounded interval of time. An episode is our general term for an activity, whether it is abstract or concrete. We assume that an algorithmic temporal planner is used that takes a planning domain, initial states and a QSP as input, and returns a plan or a set of plans if one exists, or a signal indicating that no plan exists.

We use *Executable Temporal Plans* (Temporal Plans for short) to represent the solution to a planning problem. A temporal plan contains a set of activities, which represent action sequences that can generate state trajectories to satisfy the goals specified in QSPs. We say that a plan is complete if it logically entails all goals in the QSP, and consistent if the plan itself is logically consistent, that is, all the preconditions and maintenance conditions of all activities are satisfied. A planning problem is *feasible* if a complete and consistent plan exists for it. In addition, we use

Temporal Plan Networks [22] to encode a candidate set of temporal plans that may be used to satisfy a QSP.

A planning problem is *infeasible* if no consistent temporal plan that entails all goals in the QSP exists. That is, no plan can satisfy all the goals described in the QSP. The cause of failure is the conflicts between the goals and planning domains, that is, the allowed actions and states in the plan are insufficient for satisfying the goals. For an inconsistent planning problem, there is either no complete plan that entails the QSP, or all complete plans that entail the QSP are inconsistent.

To resolve an infeasible temporal planning problem, we need to remove or change some of the goals in the QSP so that a complete and consistent plan can be generated. In this thesis, we focus on restoring the consistency of an inconsistent temporal plan by modifying some goals in the QSP. Such a modification is called a *relaxation* to a QSP, and can be applied to either goals on states that are specified by episodes, or goals on temporal relations that are specified by temporal constraints. More specifically, we focus in this thesis on schedule relaxations, which relax the users' temporal constraints, in this thesis.

In Section 2.1, we present the definition of the goal specifications in temporal planning problems using QSPs. In Section 2.2, we describe the solutions using temporal plans and TPNs. Then we discuss the causes of infeasible temporal planning problems in Section 2.3. Finally, in Section 2.4, we present relaxations as the resolutions to infeasible temporal planning problems.

2.1 Modeling User Goals using Qualitative State Plans

This section introduces a representation that captures the users' desired goals in a planning problem. In the PTS scenario, the passenger, John, propose a set of goals he would like to achieve throughout his trip. These include his requirements on time, such as *Arrive home in 80 minutes*, and requirements on the locations, like *Dinner at a sandwich restaurant*. In general, all the goals and requirements of a user can be described explicitly using a set of time evolved states and temporal constraints. The desired evolution of goal states can be represented as a *Qualitative State Plan (QSP)*:

Definition 1. *A* **Qualitative State Plan (QSP)** *is a tuple* $< \mathcal{E}, \mathcal{EPS}, \mathcal{TC}, e_{start}, e_{end} >>$ *where:*

- $\mathcal{E}$ *is a set of events. Each event* $e \in \mathcal{E}$ *can be assigned a non-negative real value, and denotes a distinguished point in time.*
- $\mathcal{EPS}$ *is a set of episodes. Each episode specifies one or more allowed state trajectories between a starting and an ending event.. They are used to represent the state constraints. An episode is a tuple* $< e_S, e_E, l, u, SC >$ *where* e_S *and* e_E *in* E *are the start and end events of possible state trajectories,* l *and* u *are lower and upper bounds on the time duration of the episode and* SC *is a set of state constraints that must be true over the duration of the episode. In this thesis, the set of state constraints* SC *is represented by a conjunction of PDDL predicates.*
- $\mathcal{TC}$ *is a set of simple temporal constraints between events E. It is used to represent the temporal constraints in the QSP. A simple temporal constraint [12] is a tuple* $< e_S, e_E, LB, UB >$ *where:* e_S *and* e_E *in* E *are the start and end events of the temporal constraint.* LB *and* UB *represent the lower and upper bounds of the duration between events* e_S *and* e_E*, where* $LB \in \mathbb{R} \cup -\infty$*,* $UB \in \mathbb{R} \cup +\infty$ *such that* $LB \leq \textsc{Time}(e_E)$ - $\textsc{Time}(e_S) \leq UB$.

- e_{start} *and* $e_{end} \in \mathcal{E}$ *are two distinct events that represents the first and last events in the QSP.*

For example, the QSP of John's trip is summarized in (Figure 2-1).

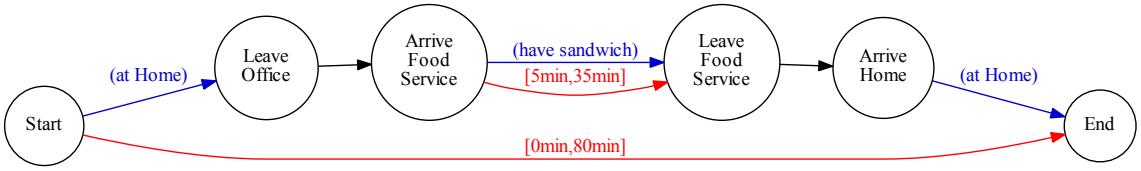


Figure 2-1: The Qualitative State Plan of John's trip

The events in the graph are represented by circles, and episodes and temporal constraints are represented by arrows. The QSP is a goal specification, comprised of two types of constraints: **state constraints** and **temporal constraints**. They are represented by episodes and simple temporal constraints. In (Figure 2-1), the episodes are represented by blue arrows with a label indicating the state constraint. A state constraint is a conjunction of propositions, where each proposition is a predicate applied to one or more variables and constraints, such as location and temperature. There are three types of state constraints that are allowed:

- Constraints on the states of the agents in a QSP, like locations, temperature and velocity.
- Instantiations of primitive PDDL operators, like movements and deformation.
- A program which can be expanded to a QSP.

For example, (Figure 2-2) shows an example of a QSP episode that constrains the location, which represents the user's requirement of not staying at the office.

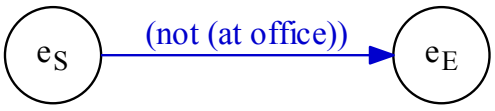


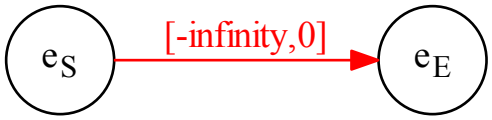
Figure 2-2: An example of episodes

In John's QSP, there are three episodes, specifying his three state constraints: the trip starts from his home (at John home), needs to include a sandwich place for dinner (have sandwich John) and finally returns home (at John home).

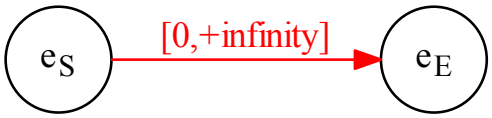
In addition, there are two temporal constraints: the dinner should last between 5 and 35 minutes, and the trip duration should be less than 80 minutes. These specify the relation that John would like to achieve between his state constraints. Temporal constraints in a QSP are described by *Simple Temporal Constraints*.

Figure 2-3: A Simple Temporal Constraint

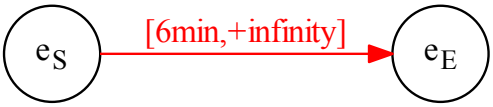
A simple temporal constraint is represented by labeled red arrows in the graph (Figure 2-3). The constraint arrow starts from the start event (e_S) and points to the end event (e_E). The lower bound of a simple temporal constraint is unconstrained if it is set to $LB = -\infty$. Similarly, its upper bound is unconstrained if $UB = +\infty$. There are several special forms of simple temporal constraints that are commonly used while describing real world scenarios (Figure 2-4):



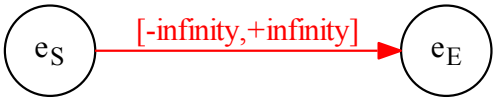
(a) e_E no later than e_S



(b) e_E no earlier than e_S



(c) e_E at least 6 mins later than e_S



(d) Constraint suspended

Figure 2-4: Special Simple Temporal Constraints

2.2 Modeling Solutions to User Goals as Temporal Plan Networks (TPNs)

This section reviews the representation of solutions to a temporal planning problem. As mentioned in the intro section, the solution to a planning problem is a temporal plan, which is a set of activities that satisfies all the state and temporal constraints in a QSP. We present two key concepts that define a valid plan in this section: *Completeness* and *Consistency*.

2.2.1 Encoding One Solution using Temporal Plans

A *Temporal Plan* is a formalism that specifies a set of activities that can satisfy all state and temporal constraints in a QSP, which is a specification of users' goals. Its form is similar to that of QSPs, but the *episodes* in a QSP are restricted to *activities*. Formally, a *Temporal Plan* is defined as:

Definition 2. *A* **Temporal Plan** *is a tuple* $< \mathcal{E}, \mathcal{ACT}, \mathcal{TC}, e_{start}, e_{end} >$ *where:*

- $\mathcal{E}$ *is a set of events. Each event* $e \in \mathcal{E}$ *is assigned a non-negative real value, and denotes a specific instant in time. This is the same concept as event in QSPs.*

- $\mathcal{ACT}$ *is a set of activities between events. It is a specialization of an episode. An activity is an episode in which its state constraint is expressed by an operator instantiation. An* **activity** *is a tuple* $< e_S, e_E, l, u, act >$*. Each activity has a start event* e_S*, an end event* e_E*, a minimal duration* l*, a maximum duration* u *and an action act. In this thesis, act is an action that represents a state transition, through its preconditions and its effect. More generally, an activity represents a state trajectory over its duration. The duration of the activity will be restricted by the temporal bounds, [l,u]. If* $u > l$*, this activity is a partial operator instantiation since the duration of this activity is flexible. Otherwise this activity is a full operator instantiation.*

- *$\mathcal{TC}$ is a set of simple temporal constraints. Like QSPs, simple temporal constraints in temporal plans specify the allowed durations between events. In addition, $\mathcal{TC}$ and $\mathcal{ACT}$ entail the temporal constraints in the QSP.*

- *e_{start} and $e_{end} \in \mathcal{E}$ are two distinct events that represent the first and last events in the temporal plan. We assume that the time assigned to e_{start}, $t_{e_{start}}$ is always 0. e_{start} and e_{end} are always connected by activities or simple temporal constraints, or a combination of both.*

Temporal plans share the same structure as qualitative state plans. The difference is that a temporal plan contains a set of activities, instead of episodes, that can satisfy the required state trajectories stated in the QSP. In other words, a QSP contains a set of goals and a temporal plan contains activities that will achieve those goals. For example, if a state constraint in a QSP imposes a transition between two locations, (at office) and (at home), then an activity, (Drive office home), may be found in the temporal plan that satisfies this constraint.

Each activity in a temporal plan implies a fully or partially instantiated state trajectory that entails some of the episodes in QSPs. This is because the start time of each activity may be fully specified or flexible, depending on the temporal constraints. If the activity has a firm start time and duration, then it will generate a set of fully instantiated state trajectories. The temporal constraints in a plan represent the temporal relations between activities, and entail the temporal constraints in its QSP.

A temporal plan is a feasible solution to a planning problem if it is *Complete* and *Consistent*. A *Complete* temporal plan logically entails all the goals, which are specified by state and temporal constraints in the QSP.

Definition 3. *[Completeness of Temporal Plans] A temporal plan $\mathcal{P}$ for a QSP $\mathcal{Q}$, is* **complete** *if the activities and temporal constraints in $\mathcal{P}$ entail $\mathcal{Q}$, $\mathcal{P} \models \mathcal{Q}$. In other words, all the state and temporal constraints in $\mathcal{Q}$ can be satisfied by $\mathcal{P}$.*

A temporal plan is **state complete** *if all the state constraints $\mathcal{EPS}$ in $\mathcal{Q}$ are entailed by all activities and temporal constraints in $\mathcal{P}$.*

A temporal plan is **temporally complete** *if all the temporal constraints $\mathcal{TC}$ in $\mathcal{Q}$, are entailed by all activities and temporal constraints in $\mathcal{P}$.*

A temporal plan is complete if it is both spatially and temporally complete.

On the other hand, even though a complete plan achieves all the goals specified in the QSP, it may not be a valid solution due to its inconsistency. The *consistency* of a temporal plan is about the consistency of the elements in it. It is necessary in order for the plan to be executed.

Definition 4. *[Consistency of Temporal Plans] A temporal plan $\mathcal{P}$ is* **consistent** *if the activities $\mathcal{ACT}$ and temporal constraints $\mathcal{TC}$ in $\mathcal{P}$ are logically consistent. In other words, no logical contradiction can be derived from $\mathcal{P}$.*

A temporal plan is **spatially consistent** *if all the activities $\mathcal{ACT} \in \mathcal{P}$, are consistent. That is, two actions do not threaten each other. This correspond is enforced through mutual exclusions.*

A temporal plan is **temporally consistent** *if all the temporal constraints $\mathcal{TC} \in \mathcal{P}$, are consistent and are satisfied by the durations of $\mathcal{ACT}$.*

The consistency of a temporal plan indicates whether it can be correctly executed. In most cases, the state and temporal consistency of a plan are coupled, since the preconditions and maintenance conditions of an activity may hold only during a certain period of time. For example, assume that drinking a bottle of soda requires two activities: *Opening the bottle* and *Drinking*. Then a temporal plan of these activities is spatially consistent if and only if the temporal constraints allow *Opening the bottle* to be executed prior to *Drinking*. In [25], a method is presented to check the spatial consistency of activities with flexibility in execution time. In this thesis, we assume that a set of temporal constraints has been introduced by the planner to guarantee that the sequence of activities satisfy the pre- and maintenance conditions of all activities at the times required.

In order to execute a temporal plan correctly, the activities of a temporal plan must be executed at proper times that satisfy all temporal constraints in the plan. The dispatch time of an activity is the same as the time assigned to its start event,

and the duration of the activity is the difference between the time assigned to its start and end events. The time that activities are executed is specified by a schedule for the temporal plan.

Definition 5. *(Schedule for a Temporal Plan) A* **consistent schedule** $\mathcal{T}$ *for a temporal plan* $\mathcal{P}$ *is a set of time assignments to all its events,* $\mathcal{E}$*, such that all the temporal constraints and activity durations in* $\mathcal{P}$ *are satisfied. Each event,* $e \in \mathcal{E}$*, is assigned a time point* t_e*. For each temporal constraint and activity duration in* $\mathcal{P}$*, the time assignments to its start and end events satisfies:* $TC_{LB} \leq t_{end} - t_{start} \leq TC_{UB}$*.* TC_{LB} *and* TC_{UB} *are the lower and upper bound of an activity duration or temporal constraint.*

For example, one temporal plan that can satisfy John's goals is shown in (Figure 2-5). He may drive to Quiznos for dinner after he leaves his office. There are three activities in this plan: 'Drive from Office to Quiznos', 'Take-out sandwich from Quiznos' and 'Drive from Quiznos to Home', represented by green arrows in the graph. The duration of each activity is different: 'Take-out Quiznos' is fully instantiated and the duration is fixed to 10 minutes, while the other activities are partially instantiated. Driving from office to Quiznos may take any time between 30 and 40 minutes. The definition of entailment between temporal plans and QSPs is presented in [25].

There is a temporal constraint that specifies the overall time requirement of the QSP: [0min, 80min]. It connects the first and last events in the temporal plan and restricts the duration of the whole trip. Similar to QSPs, temporal constraints are represented by red arrows in temporal plan graphs.

Figure 2-5: A schedule for a temporal plan

The time marked on each event shows a schedule for John's trip plan back home:

- Leave home right now (0 minute).

- Arrive at Quiznos 30 minutes from start.
- Ask for take-out and leave Quiznos 40 minutes from start.
- Arrive home 75 minutes from start.

It can be seen from the graph that all the temporal constraints and activity durations are satisfied by this consistent schedule: John spends exactly 10 minutes having dinner at Quiznos and the time assigned to two driving activities falls into the allowed durations. For any temporal plan, we can use the existence of a schedule to check its temporal consistency.

Definition 6. *[Temporally Consistent Plans] A temporal plan $\mathcal{P}$ is* **temporally consistent** *if there exists at least one consistent schedule, $\mathcal{T}$ to $\mathcal{P}$. That is, there is at least one set of time assignments to all events, $\mathcal{E}$ in $\mathcal{P}$ such that all the temporal constraints, including activity durations, are satisfied.*

For example, the temporal plan in (Figure 2-5) is temporally consistent, since it has a consistent schedule that satisfies all temporal constraints. However, if John reduces his expected arrival time from 80 minutes to 60 minutes (Figure 2-6), then no consistent schedule can be found. Such a temporal plan is not consistent, and more specifically, is defined as an *over-constrained* temporal plan.

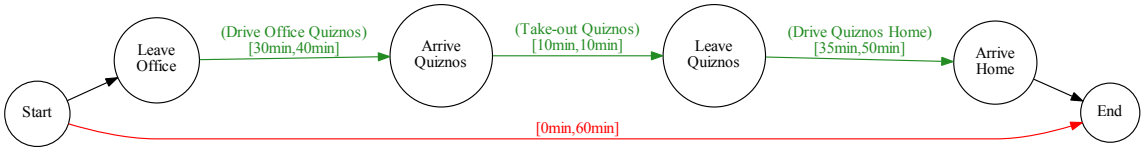


Figure 2-6: An over-constrained temporal plan

Definition 7. *[Over-constrained Temporal Plan] A temporal plan $\mathcal{P}$ is over-constrained if $\mathcal{P}$ does not have a consistent schedule. In other words, it is complete with regards to a QSP, but there are no time assignments to the events in $\mathcal{P}$ such that all its temporal constraints, including activity durations, can be satisfied.*

2.2.2 Encoding Multiple Contingent Solutions using Temporal Plan Networks

For a QSP, there might be more than one plan that is complete and consistent: several temporal plans may be found to satisfy the goals in one QSP. Usually, this is a result of two reasons:

- The goal specification in a QSP may be instantiated in different ways. For example, in John's scenario, the QSP only specifies his requirement of (Having Dinner at a Sandwich Place), but does not indicate which restaurant to go to. This episode can be instantiated with any sandwich restaurant that is on his way back home, such as 'Dinner at Subway'. Therefore, a QSP may represent multiple consistent executions if not all episodes in it are fully grounded.

- There may be multiple ways to satisfy one state constraint. For example, John may ask the planner to find a plan back home from his office. The planner identifies several different ways to commute: (Drive Office Home), (Taxi Office Home) and (Bike Office Home). A temporal plan can be generated based on each mode of commuting, hence multiple plans may be available to solve John's problem.

To represent a set of candidate plans that satisfy a QSP, we use the concept of *Temporal Plan Networks (TPN)*, a compact representation of multiple temporal plans introduced by [22].

Definition 8. *A* **Temporal Plan Network (TPN)** *is a tuple* $< \mathcal{E}, \mathcal{SP}, \mathcal{TC}, \mathcal{DE}, e_{start}, e_{end} >$ *where:*

- $\mathcal{E}$ *is a set of* **conditional** *events. Each event* $e \in \mathcal{E}$ *is a plan element that can be assigned to a specific point in time. A* **conditional** *event, e, may belong to different sub plans. e will only occur and be scheduled if any of those sub-plans are selected and executed.*

- $\mathcal{SP}$ *is a set of sub-plans. Each sub-plan is either a temporal plan, or a TPN. The start and end events of a sub-plan belongs to* $\mathcal{E}$*, that is,* e_s *and* $e_e \in \mathcal{E}$*.*

- $\mathcal{TC}$ *is a set of simple temporal constraints. Like temporal plans, simple temporal constraints in TPNs specify the allowed temporal durations between events, and entail the temporal constraints of the QSPs.*

- $\mathcal{DE}$ *is a set of decision events in the TPN, and is a subset of* $\mathcal{E}$*. A decision event, de, is an event followed by a subset of sub-plans: only one of them can be selected at a time. Its domain,* $\mathcal{DSP}$*, is the set of all sub plans whose start event is de.*

- e_{start} *and* $e_{end} \in \mathcal{E}$ *are two distinct events that represent the first and last events in the TPN. We assume that the time assigned to* e_{start}*,* $t_{e_{start}}$ *is always 0. Both events are always connected by sub plans or simple temporal constraints, or a combination of both.*

A TPN is a nested set of non-deterministic choices between alternative sub plans [23]. It is a compact representation of multiple temporal plans using choices: the activation of sub plans depends on the choices made to the decision events. In this thesis, we use TPNs to represent a combination of a (possibly incomplete) set of candidate plans that may satisfy the users' goals in a planning problem.

For example, (Figure 2-7) shows a TPN Of candidate plans for to John's trip problem. It encodes six temporal plans that may satisfy John's goals. He can have dinner at Quiznos, Subway or Cosi. At each restaurant, he has two options: take-out and dine-in. Instead of creating one choice followed by six independent temporal plans, the TPN uses nested sub plans to make the representation compact. John will only eat at Quiznos if he chooses to drive to Quiznos after he leaves the office (The Choice at event "leave office"). Otherwise, the sub plan of activities 'Drive to Quiznos', 'Dine-in' and 'Drive Home' will not be activated and executed.

In the TPN, the activities are represented by green arcs with PDDL actions and duration labels. There are four choices in the TPN: 'LeaveOffice', 'ArriveCost', 'ArriveQuiznos' and 'ArriveSubway'. They are represented by double circles in the graph.

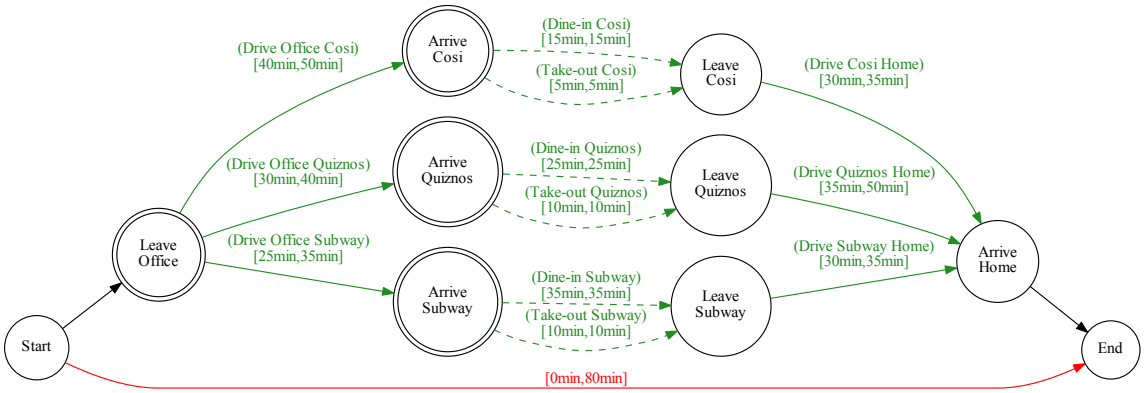


Figure 2-7: The Temporal Plan Network for John's trip

John can select one of the three restaurants to go to after leaving the office, and he can choose to dine-in or take-out when he arrives at a restaurant.

In total, the tpn encodes six candidate temporal plans for John to choose from. A temporal plan is like a TPN without any choices, therefore all activities and events are activated. To extract a temporal plan from a TPN, one may make a set of Make a set of assignments to the choices of the TPN to eliminate contingencies.

Definition 9. *(An assignment of choices in TPNs) A* **choice** *to a TPN is a pair* $< de, sp >$ *where:*

- *de is a decision event with domain* $\mathcal{DSP}$*.*
- *sp is a sub plan and* $sp \in \mathcal{DSP}$*.*

However, not all choice sets to a TPN result in a temporal plan. A set of choices is valid only if it is complete.

Definition 10. *(Assignments to TPNs) A set of assignments,* θ*, to a TPN is* **complete** *if and only if:*

- *There is no decision event that is activated by* θ *but not assigned.*
- *All decision events in* θ *must be either* **always active** *or activated by one of the choice in* θ*.*

A set of assignments to a TPN is **incomplete** *or* **partial** *if there is a decision event that is activated [14] but not assigned.*

A set of assignments to a TPN is **superfluous** *if there is a decision event that is assigned but neither activated by one of the choice nor* **always active**. *An event is* **always active** *if it is activated in all sub-plans regardless of the choices made to the decision events.*

Note that the completeness of A choice assignment is different from the completeness of temporal plan. A complete set of choices to a TPN will result in a temporal plan that supports the goals of the QSP. Such a temporal plan is called a *candidate temporal plan* of the TPN:

Definition 11. *(Candidate Temporal Plans of a TPN) A* **Candidate Temporal Plan**, $\mathcal{P}$, *of a TPN,* $\mathcal{N}$ *is a temporal plan where:*

- $\mathcal{P}$*'s events,* $\mathcal{E}$*, is a subset of the events in* $\mathcal{N}$*.*
- $\mathcal{P}$*'s sub plans,* $\mathcal{SP}$*, is a subset of the sub plans in* $\mathcal{N}$*.*
- $\mathcal{P}$*'s temporal constraints,* $\mathcal{TC}$*, is the same as the temporal constraints in* $\mathcal{N}$*.*
- $\mathcal{E}$*,* $\mathcal{SP}$ *and* $\mathcal{TC}$ *can be activated by one complete set of choices to* $\mathcal{N}$*.*

A TPN may have multiple candidate temporal plans, depending on the number of choices and the domain size of each decision events. For example, (Figure 2-8) is a candidate temporal plan of the previously mentioned TPN. It takes John to Quiznos and has join dining in at the restaurant.

Further, the solution to a QSP can be defined as a complete and consistent TPN, in which the state and temporal constraints specified in the QSP are satisfied by at least one of the candidate plans of the TPN. For example, (Figure 2-7) is a consistent TPN, and is complete with regarding to the temporal planning problem with John's goals specified in (Figure 2-1).

Definition 12. *(Complete and Consistent Temporal Plan Networks)*

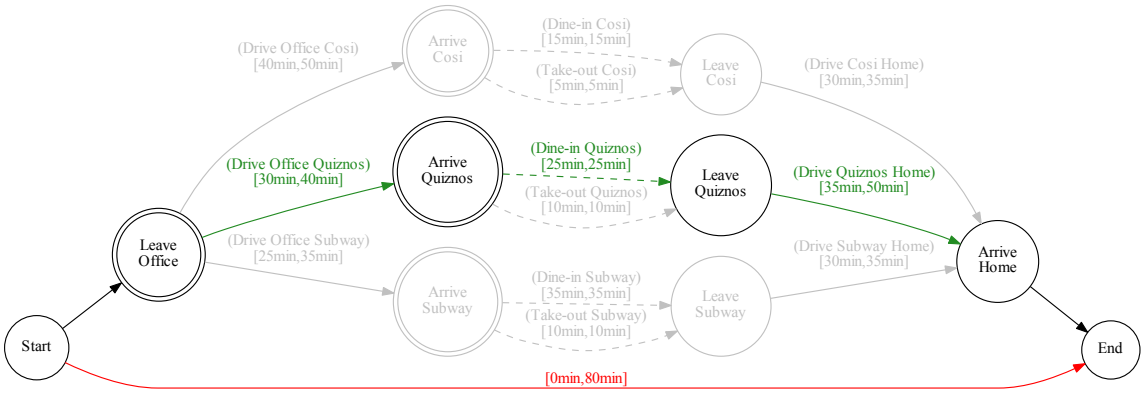


Figure 2-8: One temporal plan for John's trip

A TPN is **complete** *if at least one of its candidate temporal plans is* **complete**.

A TPN is **consistent** *if at least one of its candidate temporal plans is* **consistent**.

Finally, similar to over-constrained temporal plans, we can define over-constrained TPNs:

Definition 13. *(Over-constrained Temporal Plan Networks) A temporal plan network* $\mathcal{TPN}$ *is* **over-constrained** *if none of its candidate temporal plans is both* **complete** *and* **consistent**, *but at least one of them is an* **over-constrained** *temporal plan.*

2.3 Failure When Generating Complete and Consistent Plans

Recall that our problem of relaxing a QSP is driven by the fact that the QsP as given can't be solved. Specifically, there are some temporal planning problems for which no temporal plan can be found to satisfy all state and temporal constraints in the problem. Such a problem is called an *infeasible* temporal planning problem. In this section, we define the feasibility of temporal planning problems using temporal plans. Further, if a problem is infeasible, it indicates that there are conflicts between the goals specified in its QSP. These conflicts provide useful clues about where to relax the QSP. We discuss two possible causes of failure in this section based on the type of conflicts.

2.3.1 Defining the Feasibility of Qualitative State Plans

Given a temporal planning problem, we define its feasibility based on the solution that can be generated by a planner:

Definition 14 (Feasible QSPs)**.** *A QSP, $\mathcal{Q}$, is* **feasible** *if and only if there exists a temporal plan, $\mathcal{P}$ where:*

- *$\mathcal{P}$ is complete. It satisfies all the state and temporal constraints specified in the QSP of $\mathcal{Q}$: the state trajectories generated by the activities in $\mathcal{P}$ satisfy the state constraints, and the state and end time of the state trajectories satisfy the temporal constraints. In other words, $\mathcal{P} \models \mathcal{Q}$.*

- *$\mathcal{P}$ is consistent. There is no logical inconsistency between the activities and temporal constraints in $\mathcal{P}$. Every precondition has an action that precedes it, which produces the effect that is desired by the precondition.*

  Otherwise the QSP is said to be infeasible.

We may separate the infeasible problems into three categories:

- Given a planning problem, $\mathcal{Q}$, no complete temporal plan can be found that entails all the state and temporal constraints in $\mathcal{Q}$.
- Given a planning problem, $\mathcal{Q}$, there exists a complete temporal plans, $\mathcal{P}$, that can satisfy all the constraints in $\mathcal{Q}$. However, none of the plans in $\mathcal{P}$ is consistent.
- Given a planning problem, $\mathcal{Q}$, there is only a set of incomplete but consistent plans, $\mathcal{P}$. All plans in $\mathcal{P}$ are consistent, but cannot satisfy all the constraints in $\mathcal{Q}$.

We may further divide the second category based on the type of inconsistencies: $\mathcal{Q}$ may have complete but temporally inconsistent plans, meaning that some of the temporal constraints are violated; or complete but state inconsistent plans, meaning that some of the preconditions or maintenance conditions of activities are violated.

2.3.2 Outputs of a Planner Given an Infeasible QSP

In this subsection, we discuss the reasonable outputs of temporal planners when giving different QSPs. Throughout this thesis, we assume that there exists an algorithmic planner that can differentiate these types of infeasible problems. More specifically, given a temporal planning problem $\mathcal{Q}$, there are four possible outputs:

- a complete and consistent temporal plan, if the problem is feasible.
- a complete but inconsistent temporal plan, if the problem is infeasible.
- an incomplete but consistent temporal plan, if the problem is infeasible.
- a signal of failure, if the problem is infeasible and no complete plan and consistent plan exists.

A temporal planner will always try to give a complete and consistent temporal plan as the solution to the planning problems. However, no such plan exists if the problem is infeasible. An infeasible temporal planning problem is the result of the conflicts between the goals specified by the user and the planning domains. In other

words, what the user asks for cannot be supported by the planning domain. We list three possible outputs of a planner above if the planning problem is infeasible.

First, if the planner cannot find a consistent plan that can satisfy all the constraints in the QSP, but only a subset of the constraints, it will return an incomplete but consistent temporal plan. It indicates that the options in the planning domain are insufficient for satisfying all the state and temporal constraints of the problem, and it requires the user to supply more options in order to produce a plan.

For example, the robot in (Figure 2-9) is going to move from room A to room B. There is a door in between, but the robot does not know how to open it: the action 'open door' is not defined in its planning domain. Therefore, a planner would fail to generate a plan that can take the robot from room A and room B. This is the case where no complete plan exists given a planning problem.

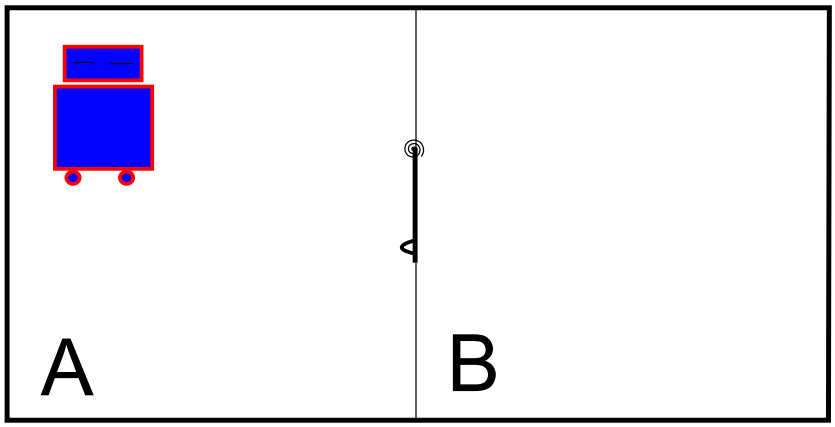


Figure 2-9: Plan failure caused by insufficient options

Second, if only complete but inconsistent temporal plans can be found with respect to the QSP of $\mathcal{Q}$, it indicates the planner can generate a set of activities that satisfies the state and temporal constraints. However:

- A. Some of the preconditions or maintenance conditions of the activities do not hold, or
- B. No schedule to these activities exists that can meet all temporal constraints.

In case A, the activities in the temporal plan cannot be executed due to the unsatisfied preconditions. This is likely to occur if a planner tries to satisfy each constraint separately and ignore the dependencies between the generated activities. In case B, the duration and sequence of the activities is incompatible with the temporal constraints in the QSP. This usually occurs when the planner is unaware of the temporal constraints in the planning process, that is, a non-temporal planner is used on temporal planning problems.

Such a plan is framed as an *over-constrained* temporal plan in Section 2.2. For example, John is late for work and would like to arrive at his office in ten minutes. However, the traffic is heavily jammed hence driving there will take at least thirty minutes (Figure 2-10). Here, the activity generated by the planner, '(Drive home office)', satisfies John's goal of arrive at his office, but the long time required for John to drive to his office would violate the temporal constraints. Under this situation, the user has to relax some of his/her goals in order to produce a feasible plan.

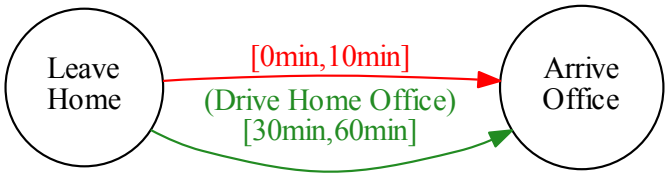


Figure 2-10: Plan failure caused by conflicts between temporal constraints and activities

Finally, the planner may return nothing but a signal of failure, meaning that it cannot satisfy even a subset of the constraints specified in the QSP. This is unlikely since it indicates that the planning domain is not related to the planning problem at all. In this thesis, we do not consider this as a reasonable output of a planner.

2.4 Resolving Infeasible Problems By Relaxing Goals

In this section, we present the resolutions to infeasible temporal planning problems. As stated in Section 2.3, no complete and consistent temporal plan can be found to infeasible problems. An infeasible problem is the result of conflicts between the constraints specified by the QSPs and the planning domains used by the planner. There are few things the user could do with the planning domain, like adding more actions, since it is usually determined by the environments. Therefore, to resolve an infeasible temporal planning problem, we have to modify some of the goals, which are specified by state and temporal constraints in its QSP.

In this section, we start by defining the QSP relaxation problem in its most general form. We characterize different categories of relaxations to a QSP. Then we describe a more specific problem of QSP relaxation and present the two specific instances addressed in this thesis. We present the position of our work in the road map as well as open problems for future research.

2.4.1 The Relaxation Problems for QSP

In this thesis, we focus on QSPs that are infeasible because one or more of its candidate plans is over constrained. We achieve QSP feasibility through a relaxation of the temporal constraints of the qsp that makes consistent one or more of the candidate temporal plans. As presented in Section 2.2, an over-constrained temporal plan is complete but temporally inconsistent. This could be generated by a planner that is not aware of temporal constraints. Given an infeasible planning problem and its over-constrained temporal plans, we can restore its consistency by modifying some episodes and temporal constraints in the QSP to enable a complete and consistent plan. Such a modification is called a *Relaxation* to a QSP:

Definition 15. *[Relaxations to QSP] Given a QSP $\mathcal{Q}$ with a set of complete but inconsistent temporal plans to it, TPN, a* **Relaxation** *is a pair $< \mathcal{E}, \mathcal{T} >$, where $\mathcal{E}$ is a set of state constraints, and $\mathcal{T}$ is a set of temporal constraints. In addition,*

- *$\mathcal{E} \models \mathcal{E}'$, where $\mathcal{E}'$ is the state constraints in $\mathcal{Q}$.*

- $\mathcal{T} \models \mathcal{T}'$, *where* $\mathcal{T}'$ *is the temporal constraints in* $\mathcal{Q}$.

such that at least one candidate plan in TPN *is now consistent and satisfies* $\mathcal{E}' \setminus \mathcal{E}$ *and* $\mathcal{T}' \setminus \mathcal{T}$.

Here, TPN represents a set of temporal plans to the planning problem. Note that this relaxation does not need to rely upon TPNs. TPN can be replaced with a set of candidate plans that are considered by a planner. Given an infeasible QSP, we would like to find a relaxed set of goals, $\mathcal{E}$ and $\mathcal{T}$, that entails the original goals specified by the QSP while enabling a complete and consistent plan to be generated. The problem of generating relaxations to an infeasible QSP is defined as *Relaxation Problems*:

Definition 16. *(Relaxation Problems) Given an infeasible temporal planning problem,* $\mathcal{Q}$. *A* **Relaxation Problem** *is the problem of generating relaxations,* $\mathcal{E}$ *and* $\mathcal{T}$ *for* $\mathcal{Q}$, *such that a complete and consistent plan that satisfies* $\mathcal{E}$ *and* $\mathcal{T}$ *can be generated.*

In other words, to resolve an over-constrained plan, a relaxation modifies some of the state and temporal constraints in its QSP. As stated in the definition, there are two types of relaxations: 1) Modifying state constraints 2) Modifying temporal constraints. This thesis only discusses the schedule relaxations, in which temporal constraints of a QSP are modified in order to enable a complete and consistent plan from an initially inconsistent planning problem.

2.4.2 Temporal Relaxation for QSP

Such a relaxation is called a *Temporal Relaxation*. Formally, a *Temporal Relaxation* to a QSP is defined as:

Definition 17. *(Temporal Relaxation to a QSP) A* **Temporal Relaxation**, $\mathcal{TR}$, *to a QSP* $\mathcal{Q}$ *is a set of temporal constraints modified from the subset of the temporal objectives* $\mathcal{T}$ *where:*

- $\mathcal{TR} \models \mathcal{T}$.

- *Replacing $\mathcal{T}$ in $\mathcal{Q}$ with $\mathcal{TR}$ enables a complete and consistent temporal plan to be generated.*

That is, the temporal relaxation modifies some of the temporal constraints in the QSPs. As presented in Section 2.1, we use simple temporal constraints to represent the users' requirements on temporal relaxations, and each simple temporal constraint restricts the times assigned to its start and end events. Therefore, relaxing a temporal constraint is in fact modifying the temporal bound of it. In this thesis, we introduce two restrictive forms of temporal relaxations for the temporal constraints in a QSP: discrete relaxations and continuous relaxations. First, a temporal relaxation may simply remove some of the temporal constraints in the QSP, generating a *Discrete Temporal Relaxation*:

Definition 18. *(Discrete Temporal Relaxation) A* **Discrete Temporal Relaxation**, *$\mathcal{DT}$, to an over-constrained QSP is a temporal relaxation in which a set of temporal constraints $\mathcal{DTR}$ is modified where:*

- *$\mathcal{DTR} \subseteq \mathcal{T}$. $\mathcal{T}$ is the set of temporal constraints in the QSP.*
- *$\mathcal{T} = \mathcal{T} \setminus \mathcal{DTR}$*

For example, (Figure 2-11) shows an over-constrained temporal plan in which the duration of activities is inconsistent with the temporal constraint, '[0min,60min]'. If this temporal constraint is removed, a schedule to all of the events can be found and hence making the plan consistent.

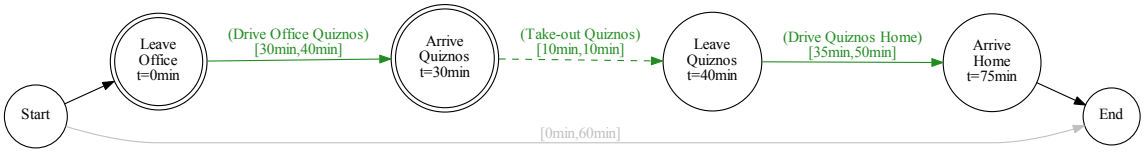


Figure 2-11: A discrete temporal relaxation to a John's trip

The discrete temporal relaxation takes an all-or-nothing approach in which temporal constraints are either preserved or removed. For example, a student who realizes

that he cannot complete the problem set on time decides not to do it. However, a better solution would be to ask for an extension. By introducing the concept of continuous temporal relaxation, a weakened version of the temporal constraints can be preserved in the relaxed problem. Therefore, the perturbations introduced by the relaxations to the user's goals can be minimized.

Definition 19. *(Continuous Relaxation to Temporal Constraints) A* **Continuous Relaxation**, $\mathcal{C}_t$, *to a simple temporal constraint, t, is a tuple* $< LB', UB' >$ *where:*

- LB' *and* UB' *are the relaxed temporal bounds of* LB, UB *in* t, *where* $LB' \leq LB$ *and* $UB \leq UB'$.

Definition 20. *(Continuous Temporal Relaxation for a Temporal Planning Problem) A* **Continuous Temporal Relaxation**, $\mathcal{CTR}$, *to a QSP with a set of infeasible candidate plans,* $\mathcal{P}$, *is a set of continuous relaxations to the temporal constraints of the QSP where:*

- $\mathcal{CTR} \models \mathcal{T}$. $\mathcal{T}$ *is the set of temporal constraints in the QSP.*
- *Each* $t_{cr} \in \mathcal{CTR}$, *is a simple temporal constraint continuously relaxed from a simple temporal constraint* $t_g \in \mathcal{P}$.
- *Replacing all* $t_g \in \mathcal{P}$ *with* $t_{cr} \in \mathcal{CTR}$ *enables a complete and consistent temporal plan to be generated with regard to the QSP.*

The continuous relaxation is a generalization of the discrete relaxation: discrete relaxations can be viewed as relaxing the temporal constraints to $[-\infty, +\infty]$. For example, (Figure 2-12) shows a continuous relaxation to John's QSP: a consistent plan exists if the temporal constraint of his trip duration can be extended from 60 minutes to 75 minutes.

In summary, given an infeasible planning problem with its over-constrained plans, we may resolve it by relaxing the temporal constraints. More specifically, we resolve the inconsistencies between the activities and the temporal constraints by generating relaxations to the temporal constraints. There are two types of temporal relaxations: discrete relaxations that completely suspend the constraints, and continuous

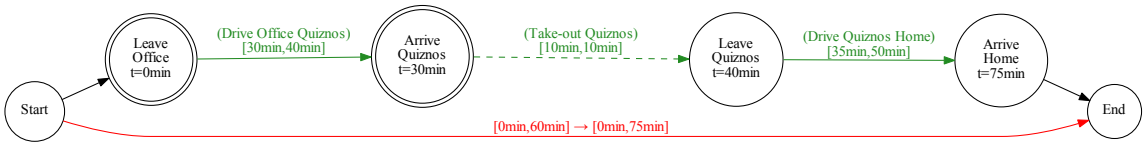


Figure 2-12: A continuous temporal relaxation to a John's trip

relaxations that partially relax the constraints by modifying their temporal bounds. Compared to discrete relaxations, continuous relaxations preserve plan elements to the maximum and minimize the perturbations to the users' goals. However, knowing discrete relaxation is a stepping stone to the continuous relaxation. We will present the algorithm that generates discrete relaxations first in Chapter 3, and then present the continuous relaxation algorithm build on the discrete one in Chapter 4.

2.5 Chapter Summary

This chapter defines several important concepts for this thesis. In a temporal planning problem, the users' goals are specified by state and temporal constraints and encoded in *Qualitative State Plans*. The solution to a planning problem is a *Complete and Consistent Temporal Plan*, which contains a set of activities that entails the goal evolution in its QSP. If there are multiple solutions, we can use a *Temporal Plan Network* to encode them compactly.

A temporal planning problem is *feasible* if such a complete and consistent plan exists. A planning problem is *infeasible* if the planner fails to return a complete plan, or returns a complete but inconsistent plan. We say that a temporal plan is *over-constrained* if it is complete, in that it entails each of its goal states, but is inconsistent. An *over-constrained* temporal plan is complete in that it entails all the goals in the QSP, but is temporally inconsistent.

Infeasible planning problems are the result of conflicting goals and the options in planning domains, and different type of conflicts lead to different type of failures. This thesis focuses on problems whose temporal plans are *over-constrained*, which is a result of conflicting activity durations and temporal constraints. We can resolve the problem by generating *temporal relaxations*, which modify the temporal constraints to enable a consistent plan to be generated.

Two useful relaxations to *over-constrained* temporal plans are *Discrete Temporal Relaxations* and *Continuous Temporal Relaxations*, in which the temporal constraints in the QSPs are modified. Discrete temporal relaxations suspend a set of the temporal constraints completely, while continuous temporal relaxations is its generalization that preserves all the temporal constraints. Continuous temporal relaxations only adjust the temporal bounds of temporal constraints in order to restore consistency.

Chapter 3

Relaxing Inconsistent Temporal Problems using Conflict-directed Diagnosis

In this chapter we introduce a novel approach that enumerates preferred minimal relaxations for over-constrained goals, which can be represented by over-constrained temporal plans. We begin by mapping the temporal plans to Optimal Conditional Simple Temporal Networks (OCSTN), a CSP formulation with conditional and linear constraints. Our algorithm, called *Best-first Conflict-directed Relaxation (BCDR)*, can resolve inconsistent OCSTNs efficiently using pruning techniques adopted by CSP solvers and conflict-directed enumeration algorithms. BCDR is the core technique within Uhura that supports collaborative plan diagnosis.

In the context of TPNs, BCDR resolves over-constrained problems by relaxing the temporal constraints, which can be viewed as a schedule relaxation when referring to temporal plans. It draws insights from other constraint satisfaction solvers like CD-A* [37] and Dualize & Advance [4], and generates discrete relaxations that resolve inconsistent OCSTNs by suspending temporal constraints. BCDR can also generate continuous relaxations, which significantly reduce the perturbations to the users' original plans: constraints are not suspended but only minimally modified. This chapter focuses on enumerating discrete relaxations. It is a stepping stone to

enumerate continuous relaxations, which will be presented in Chapter 4.

Compared to current relaxation methods, BCDR is innovative in two ways:

- BCDR views over-subscription and its corresponding relaxation as an instance of consistency-based diagnosis [10, 9]. It avoids redundant solutions by introducing the concept of *minimal temporal relaxations*. For example, if $\mathcal{A}$ is a minimal temporal relaxation and $\mathcal{B}$ is a superset of $\mathcal{A}$, then $\mathcal{B}$ is not minimal and will not be generated. Assuming that users prefer to relax as few constraints as possible in any situation, then all proper supersets of $\mathcal{A}$ are less preferable and unnecessary for the enumeration. This is in contrast to [27], which enumerates full relaxations for over-constrained problems.

- BCDR efficiently enumerates only minimal relaxations in best-first order. It scales relaxation to large problems by unifying algorithms for fast enumeration based on conflict-directed A*, with algorithms for detecting minimal relaxation from constraint explanations. This is in contrast to [4], which does not prioritize the relaxations.

Recall from the Chapter 1, BCDR has been implemented as part of Uhura for a robotic air taxi, the Personal Transportation System, to support the collaborative diagnosis of over-subscribed plans. Through a rich set of benchmark tests, we demonstrate that BCDR achieves nearly two orders of magnitude improvement in run-time performance of temporal relaxations compared with previous relaxation algorithms. The benchmark result is presented in Chapter 5.

Section 3.1 presents the mapping between TPNs and OCSTNs. We demonstrate the equivalence of TPN and OCSTN consistency that enables us to resolve over-subscribed TPNs as inconsistent OCSTNs. Section 3.2 presents the problem statements for BCDR in the context of resolving inconsistent OCSTNs. We define discrete relaxation problems for OCSTNs based on the relaxation problems of TPNs. We present an overview of the BCDR algorithm in Section 3.3. BCDR contains four major steps: consistency checking, conflict extraction, candidate generation and selection. We elaborate and present the details of each step in Section 3.4-3.7.

3.1 Modeling Temporal Plan Networks using Optimal Conditional Simple Temporal Networks (OCSTNs)

As presented in Chapter 2, we use *Temporal Plan Networks* [22] to encode a set of plans that can satisfy the users' requirements. A TPN is a compact representation of multiple temporal plans composed by decision events. Within each temporal plan, there is a set of timed activities that can satisfy the state and temporal constraints specified by the user.

In this section, we review the mapping between a TPN and an Optimal Conditional Simple Temporal Network (OCSTN, [14]). In an OCSTN, all continuous variables have real-valued domains with set-bounded constraints and all discrete variables determine the activation of these constraints. An OCSTN is an extension to a CSP, which adds conditional and linear constraints, hence enabling the use of constraint-based techniques in BCDR (Section 3.3). Note that an OCSTN simply lacks the specification of activities and operator instantiation, but this does not matter for the purpose of scheduling. We demonstrate that each TPN has its unique corresponding OCSTN, and the temporal consistency of the TPN is the same as its equivalent OCSTN.

3.1.1 The Definition of an OCSTN

An OCSTN [14] is a hybrid CSP formalism that combines the Optimal Constraint Satisfaction Problem (OCSP) and the Conditional Simple Temporal Network (CSTN) [35], Which are inn turn composed of Conditional CSPs [17] and STNs [12]. An OCSTN has two important features:

- Conditional: it uses both conditional variables and constraints. Decision variables are specified to capture the disjunctive relations between sub-problems. A decision is specified by a guard, which activates corresponding decision variables and constraints that rely on this guard.

- Optimal: It uses a utility function over decision variables and constraints to capture the users' preference towards different sub-problems and relaxations.

- Temporal: Its constraints are simple temporal constraints.

Both features are necessary to encode a TPN: a TPN is a compact representation of multiple candidate temporal plans with decision events, and the user can specify his/her preference using real-valued utility functions. The definition of OCSTN extends the Simple Temporal Network ([12]), and is presented in Definition 21:

Definition 21. *(Optimal Conditional STN) An OCSTN is defined as a 6-tuple* $< P, P_i, V, E, GC, f(P) >$. *Where,*

- P *is a set of discrete decision variables. Each decision variable* p_i *is a tuple* $< p_i, guard >$. *It may have an associated guard condition guard, which is an assignment to a decision variable. A variable is active if it has no guard or if the guard is satisfied by the current assignment to decision variables.*

- $P_i \subseteq P$ *represents the decision variables that are always active.*

- $v_i \in V$ *is the domain of a decision variable* $p_i \in P$.

- E *is a set of events whose domains are real-valued time points.*

- GC *is a set of guarded simple temporal constraints. Each guarded simple temporal constraint is a 5-tuple,* $< e_S, e_E, LB, UB, guard >$. $e_S, e_E \in E$ *are the start and end events.* $LB, UB \in \mathbb{R}$ *are the lower and upper bounds. guard is an assignment,* p_i, v_{ik}, *to a decision variable* $p_i \in P$.

- $f(P)$ *is a multi-attribute utility function that sums up the utility values of all assignments made to* P. *This is accomplished by computing the sum of the cost of each individual assignment:* $f(A) = \sum_i Cost(a_i : p_i = v_{ik})$.

Intuitively, we make decisions by assigning values to decision variables. These assignments are called *guards*. An assignment, which represents a decision, can activate other decision variables and temporal constraints that have been guarded by this assignment.

Definition 22. *(Assignment) An assignment is a pair* $< p, v_{ik} >$ *where:*

- $p \in P$ *is a decision variable in the OCSTN.*
- $v_{ik} \in v_i$ *is a value in the domain of decision variable* p*.*
- *A guard is a single assignment to a decision variable.*

When a decision variable is active, it either has no guard or its guard is satisfied. Such a guard may activate other decision variables, if this assignment is identical to their *guard*. On the other hand, no assignment should be made to a decision variable that is not active. Note that an OCSTN has two sets of variables. The continuous variables E have domain $\mathbb{R}$ and are constrained by the guarded STN. The discrete variables P, also called decision variables, have a finite domain.

Similar to the selection of a candidate temporal plan from a TPN, by choosing a proper set of assignments from the OCSTN, we can activate a subset of the guarded constraints and instantiated the OCSTN as a regular, unguarded STN. Such a network is called a component Simple Temporal Network of the OCSTN:

Definition 23. *(Simple Temporal Network) A* **Simple Temporal Network (STN)** *is a pair,* $< \mathcal{E}, \mathcal{C} >$ *where:*

- $\mathcal{E}$ *is a set of variables. Each variable* e_i *represents a time point and has a continuous domain,* $\mathbb{R}$*.*
- $\mathcal{C}$ *is a set of simple temporal constraints between the variables. Each constraint,* c_j*, imposes a temporal requirement between the assignments of two variables in* E*. In other words, if there is a simple temporal constraint,* c_i*, between variables* e_i *and* e_j*, the time difference between these two variables,* $e_i - e_j$*, must fall into the time interval* $l_i \leq e_i - e_j \leq u_i$*.*

Given an OCSTN, we would like to find a set of assignments that grounds the OCSTN into one component STN. Such a set is called a *complete* set of assignments to this OCSTN. Intuitively, for a set of assignments, we would like it to eliminate all the disjunctions in the OCSTN, which implies that all active decision variables must be assigned. Formally, a *complete* set of assignments is defined as the following:

Definition 24. *(Complete set of assignments to an OCSTN) A set of assignments, A, to an OCSTN is complete if and only if it satisfies three conditions:*

- *For an assignment* $< p_i, v_{ik} >$ *in A, the decision variable* p_i *must be active either through another guard in A, or defined as always active.*
- *There is no decision variable* p_j *that is activated but not assigned.*
- *There is no conflicting assignment in A. That is, two assignments that are associated with the same decision variable.*

The OCSTN formalism is useful in that it satisfies all our needs of encoding TPNs: the decision events, temporal relaxations and user preferences of a TPN are well preserved in its OCSTN encoding. The solution to an OCSTN must be a set of assignments that grounds it into a STN, which maps the temporal plan grounded from a TPN, too.

3.1.2 OCSTN Consistency

Remember that previously in Chapter 2 we talked about checking consistency of a TPN: a TPN is consistent if one of its candidate temporal plans is consistent, meaning that the durations of the activities in the plan satisfy all the temporal constraints. We define the consistency of OCSTN in a similar manner: the OCSTN is consistent if one of its grounded STNs is consistent.

Formally, if there exists a set of assignments to an OCSTN that is complete and all active temporal constraints can be satisfied, then the set of assignments is said to be a consistent solution to that OCSTN.

Definition 25. *(Consistent Solutions to an OCSTN) A consistent solution Sol to an OCSTN is a complete set of assignments where all active simple temporal constraints $gc_i \in GC$ can be satisfied. That is, there is a schedule T to all events E in the component STN that obeys all active constraints.*

There might be multiple consistent solutions to an OCSTN. Using the utility function $f(P)$ in the OCSTN, we may select one solution with the best utility value. Such a solution is called the *optimal solution* to the OCSTN.

Definition 26. *(Optimal Solutions to an OCSTN) An optimal solution, OpSol to an OCSTN is one of its consistent solutions $OpSol \in Sols$ where:*

$OpSol = arg\min_{s \in Sols} f(s)$ *s.t.* $\forall gc_i \in ActiveGC$ *is satisfied.*

An optimal solution to an OCSTN has the best utility value among all consistent solutions. There might be multiple optimal solutions that have the same utility value. On the other hand, if a set of assignments is complete but not temporally consistent, then it is said to be a *conflict* for its OCSTN. Intuitively, conflicts can be interpreted as the "cause of failure", which leads to a set of active temporal constraints that cannot be satisfied.

Definition 27. *(Conflicts of an OCSTN) A conflict, Cfl, to an OCSTN is a complete set of assignments where all activated simple temporal constraints $gc_i \in GC$ can not be satisfied at the same time.*

Furthermore, we can define the minimal inconsistent subset of a conflict in an inconsistent OCSTN as a *minimal conflict*.

Definition 28. *(Minimal Conflicts of an OCSTN) A minimal conflict, $MinCfl$, to an OCSTN is a complete set of assignments and an inconsistent set of temporal constraints activated by the assignments, GC'. In addition, if any constraint $gc_i \in GC'$ is suspended, $GC' \setminus gc_i$ becomes consistent.*

Minimal conflicts are the "core cause of failure". If one can detect and resolve all minimal conflicts in an OCSTN, the consistency of it can then be restored. Finally,

we define the consistency of an OCSTN based on the complete sets of assignments and conflict we just introduced.

Definition 29. *(Consistent and Inconsistent OCSTNs)*

- *An OCSTN is consistent if and only if it has at least one consistent solution. That is, among all its complete sets of assignments, there is at least one set that is temporally consistent.*
- *An OCSTN is inconsistent if and only if it does not have a consistent solution. In other words, all complete set of assignments to it are conflicts.*

3.1.3 Encoding a TPN using an OCSTN

Now, we are going to present the connection between TPNs and OCSTNs. Remember that there are two motivations for us to use OCSTNs to encode the TPNs: First, OCSTN is structurally similar to the TPN so that we can preserve all the necessary features; Second, OCSTN is a CSP-based formalism that enables us to use efficient constraint-based search techniques. We will be focusing on the first motivation in this section, and leave the second motivation for Section 3.3.

In [14], a mapping between Optimal Conditional CSP (OCCSP) was introduced to encode TPNs. However, the OCCSP formalism is not compact for relaxation problems in that it encodes the disjunctive episodes of a TPN as domain values for decision variables: making an assignment to a decision variable is equivalent to selecting a set of temporal constraints. Relaxing an over-constrained OCCSP would be adding a domain value, which contains all but suspended temporal constraints, to a decision variable. OCSTN is a more compact encoding for relaxation problems in that a relaxation can simply be represented by a suspended temporal constraint, regardless of the decision variables.

Intuitively, we make the connection between OCSTNs and TPNs through the mapping of decision events and decision variables, guards, episodes and temporal constraints. Recall that the decision events in TPNs encodes different sub-plans.

Similarly, by choosing a set of assignments to apply to the decision variables in an OCSTN, some of its guarded temporal constraints will be activated and ground the OCSTN into a component STN that corresponds to a sub-plan in the TPN. In addition, we made an assumption in Section 2.4 that we only consider the schedule relaxation of *over-constrained* TPNs in this thesis. Therefore, we only preserve the temporal information in the encoding of TPNs while mapping the TPN episodes into OCSTN constraints.

For example, to map John's trip plan TPN (Figure 3-1) to an OCSTN (Figure 3-2), all episodes are mapped to guarded temporal constraints. Each conditional constraint is guarded with the decision required to activate it, like "ArriveCosi:(Dine-in Cosi) [15min,15min]". The activation of this constraint depends on the assignment made to decision variable "ArriveCosi": it will be respected only if "ArriveCosi" is assigned (Dine-in Cosi) instead of (Take-out Cosi).

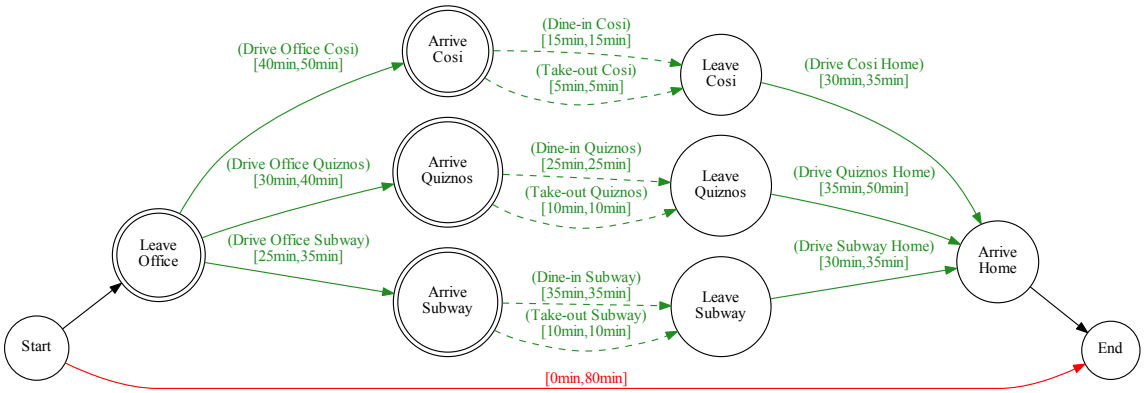


Figure 3-1: John's trip modeled as a Temporal Plan Network

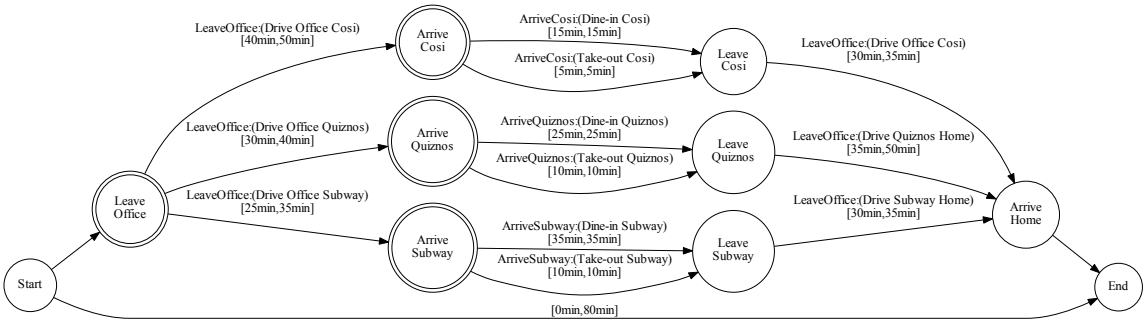


Figure 3-2: John's trip modeled as an Optimal Conditional Simple Temporal Network

Formally, we define the encoding between TPN and OCSTN as the following.

Definition 30. *(OCSTN Encoding of a TPN) An OCSTN encoding of a TPN, $\mathcal{P}$, is a 7-tuple $< P, P_i, V, E, GC, RGC, f(P) >$ where:*

- *P is a set of decision variables corresponding to the decision events in $\mathcal{P}$.*
- *P_i is a set of decision variables that are always active. It is used to represent the decision events in $\mathcal{P}$ that do not depend on choices made to any other decision events.*
- *$V = v_1, v_2, ..., v_i$ represents the domain of each decision variable $p_i \in P$. Each domain value $v_{ik} \in v_i$ corresponds to a choice in a decision event of $\mathcal{P}$.*
- *E is the set of events in $\mathcal{P}$. Each event can be assigned a real-valued time point.*
- *GC is the set of guarded simple temporal constraints. The guard is used to indicate the choice required to activate this constraint in $\mathcal{P}$. For constraints that are always active, their guards are empty.*
- *RGC is a subset of GC which represents the simple temporal constraints that can be relaxed to restore temporal consistency without violating the completeness and consistency of the TPN.*
- *$f(P)$ is the utility function that maps a set of assignments to a real value number. $f(P)$ is defined for each choice in the decision events of (P) such that a utility value can be computed for any combinations of choices.*

Finally, we present the equivalence of OCSTN and TPN consistency. In this thesis, we focus on the temporal consistency and relaxations of temporal plans. We start the proof with the equivalence between the temporal consistency temporal plans and STNs, and then expand to TPNs and OCSTNs.

Theorem 1. *A temporal plan, $\mathcal{P}$, is temporally consistent* **if and only if** *its equivalent STN is consistent.*

Proof. [Proof by contradiction]

Given a temporal plan, $\mathcal{P}$, and its equivalent STN, $\mathcal{S}$, assume that $\mathcal{S}$ is consistent but $\mathcal{P}$ is temporally inconsistent.

- If $\mathcal{S}$ is consistent, then there exists a set of time assignments TAs to all variables $\mathcal{V}$ in $\mathcal{S}$ such that all the simple temporal constraints $\mathcal{C}$ are satisfied.
- We can construct a schedule SC to $\mathcal{P}$ that is identical to TAs but with all the assignments made to the events in $\mathcal{P}$.
- Given that all the constraints in $\mathcal{S}$ are mapped from the activities and temporal constraints in $\mathcal{P}$, if TAs satisfies $\mathcal{C} \in \mathcal{S}$, then SC can satisfy $\mathcal{ACT}$ and $\mathcal{TC} \in \mathcal{P}$.

  Hence $\mathcal{P}$ is temporally consistent. the assumption does not hold.

□

We can prove the other direction using the same approach. Further, the theorem can be extended to TPNs and OCSTNs: if any of the candidate temporal plan in a TPN is temporally consistent, its equivalent OCSTN must have a component STN that is consistent. Therefore, we conclude that both the TPN and the OCSTN are consistent. Given a TPN, we can determine its consistency by encoding it into an OCSTN and checking the consistency of the OCSTN.

Theorem 2. *A TPN, $\mathcal{TPN}$, is consistent* **if and only if** *its equivalent OCSTN is consistent.*

3.2 Discrete Relaxation Problems

In this section, we define the problem of generating discrete relaxations for inconsistent OCSTNs. The goal is to find a set of **preferred** and **minimal** relaxations that can resolve the conflicts in an inconsistent OCSTN so that a consistent solution can be found.

3.2.1 Discrete Relaxations for OCSTNs

As presented in Chapter 2, a discrete relaxation for an over-subscribed TPN is a set of simple temporal constraints whose suspension makes the TPN temporally consistent. The discrete relaxation for an OCSTN can be defined in a similar manner.

Definition 31. *(Discrete Relaxation for an inconsistent OCSTN) A discrete relaxation DR to an OCSTN $\mathcal{P}$ is a set of simple temporal constraints where:*

- *DR is a subset of the simple temporal constraints in the OCSTN, $DR \subseteq RGC$, that can be relaxed without violating the completeness and consistency of its equivalent TPN.*
- *Removing DR from the OCSTN, that is, $GC = GC \setminus DR$, restores the consistency of $\mathcal{S}$.*

Recall that the utility function in an OCSTN only maps the assignment to a number. However, to compare two discrete relaxations for an OCSTN, the preference model should specify the user preferences over the suspension of different temporal constraints, as well as the preference between different outcomes of the decision events. More specifically, we define the requirements of the preference model as follows:

Definition 32. *(Preference Models over Discrete Relaxations) A preference model, $\mathcal{PM}$, for capturing the users' preferences over the discrete relaxations for an inconsistent OCSTN, $\mathcal{P}$, must satisfy the following guidelines:*

- *Domain: $\mathcal{PM}$ is defined over the complete domain of the events of $\mathcal{P}$. That is, $\mathcal{PM}$ can be used to evaluate any relaxation for any temporal constraint in*

$\mathcal{P}$, including the combinations of relaxations and choices made to the decision variables in $\mathcal{P}$. Given a relaxation $\mathcal{R}$, the result of the evaluation should be a real value that can be used for comparison, or the preferred relaxation among two or more relaxations.

- *Comparison: Given any two discrete relaxations, DR and DR', to $\mathcal{P}$, $\mathcal{PM}$ can be used to choose the one that is more preferred by the user, if not equally preferred.*

- *Evaluation: For any temporal constraint in $\mathcal{P}$, the user always prefers to preserve the constraint rather than relax it.*

3.2.2 Minimal Discrete Relaxations for OCSTNs

The number of all possible relaxations for an inconsistent OCSTN is exponential in terms of the constraints. Therefore, it would be computationally prohibitive to iterate through all of them to find the best one. Given an inconsistent OCSTN, we would like to generate the *minimal* discrete relaxation, a compact representation of all relaxations. By using the minimal relaxation, we can reduce the number of results by several orders of magnitude and speed up the enumeration process. In this section, we define the Minimal Discrete Relaxation for OCSTNs based on the discrete relaxations for TPNs presented in Chapter 2.

Definition 33. *(Minimal Discrete Relaxation) A* **Minimal Discrete Relaxation**, *MDR, for an inconsistent OCSTN, $\mathcal{P}$, is a set of simple temporal constraints where:*

- *$MDR \subseteq RGC$. RGC is the set of relaxable simple temporal constraints in $\mathcal{P}$.*

- *Suspending MDR from $\mathcal{P}$, that is, $GC = GC \setminus MDR$, restores the consistency of $\mathcal{S}$.*

- *Given a proper subset of $MDR' \subset MDR$, $GC = GC \setminus MDR'$ cannot restore the consistency of $\mathcal{S}$.*

A minimal discrete relaxation is *minimal* in that none of its subsets can restore the consistency of the inconsistent OCSTN. In other words, we do not need to consider the proper supersets of any discrete relaxations. This is the key concept that reduces the number of results generated by BCDR. For example, if we know that John's problem can be resolved by removing his temporal constraint on the trip duration, then we will not ask him to do anything beyond suspending this one constraint, like suspending both temporal constraints of trip duration and dinner time.

3.2.3 The Discrete Relaxation Problem for an OCSTN

Finally, we define the discrete relaxation problem of inconsistent conditional OCSTN:

Definition 34. *(Preferred Minimal Discrete Relaxation Problem) Given an inconsistent OCSTN $\mathcal{P}$ and a user preference model $\mathcal{UPM}$, a* **discrete Relaxation Problem** *is a problem of finding a discrete relaxation, $\mathcal{DR}$, such that three conditions hold:*

- *$\mathcal{DR}$ suspends a set of relaxable simple temporal constraints in $\mathcal{P}$ and makes $\mathcal{P}$* **consistent**.
- *$\mathcal{DR}$ is a* **minimal** *discrete relaxation.*
- *$\mathcal{DR}$ is the most* **preferred** *minimal discrete relaxation. That is, according to $\mathcal{UPM}$, $\mathcal{DR}$ is preferred to any other minimal relaxation.*

In summary, a discrete relaxation problem is composed of an inconsistent OCSTN and a preference model over its constraints and decision variables. The preference model can be used to compare two relaxations with different constraint suspensions. The desired solution to the problem is the most preferred minimal discrete relaxation. It is the most preferred discrete relaxation according to the preference model and is minimal in that we cannot simplify it by reducing any suspended constraint.

3.3 Enumerating Temporal Relaxations using Best-first Conflict-Directed Relaxation (BCDR)

If an OCSTN is inconsistent, we may apply a discrete relaxation that suspends some temporal constraints and makes the constraint network consistent. In this section, we present our *Best-first Conflict-directed Relaxation* algorithm, the core algorithm in Uhura that generates preferred and minimal discrete relaxations for inconsistent OCSTNs.

BCDR solves discrete relaxation problems by detecting conflicts in the OCSTN and generating preferred minimal temporal relaxations that restores consistency. BCDR combines ideas from the Conflict-directed A* algorithm (CD-A*) [37] and the minimal relaxation the Dualize & Advance algorithm (DAA) [4]. It uses the technique in DAA to explore the space of minimal discrete relaxations and the best-first conflict-directed technique in CD-A* to guide the search.

CD-A* is an Optimal CSP solver and was originally developed to enumerate likely diagnoses for hardware failures. It supports a domain independent theory of model-based diagnosis. This diagnosis process is framed as a form of resolving inconsistent finite domain constraint satisfaction problems. CD-A* uses conflicts detected in the problem to guide its best-first enumeration process. It exploits the duality between conflicts and minimal diagnoses, which was first noted by [10], and enumerates solutions that resolves known conflicts in best-first order.

DAA was designed to detect minimal conflicts in infeasible constraint satisfaction problems. It incrementally generates minimal conflicts and relaxations using the duality between them. Similar techniques are introduced in the *General Diagnosis Engine (GDE)* and *Sherlock* [10, 11]. This helps DAA reduce the search space and keeps the results compact. DAA also specifies the termination condition for the incremental minimal conflicts and relaxations enumeration process. The unification of CD-A* and DAA enables BCDR to (1) generate compact and parsimonious relaxations; (2) response to user queries quickly; (3) produce user preferred resolutions.

In this section, we describe the BCDR method top down, first describing the

outer loop of the method, then dive into the details of its functions. We begin by presenting an overview of BCDR. Then we describe the background and related work of conflict-directed search methods in Section 3.4. Its four major procedures: consistency checking, conflict detection, conflict resolution and the selection of preferred candidate relaxations will be discussed in Section 3.5-3.7.

In the development of BCDR, we leverage off the best-first enumeration of CD-A* and the incremental conflict detection of DAA. Both CD-A* and DAA use conflict-directed techniques: it uses conflicts to incrementally generate new candidate relaxations and discover new conflicts by testing these candidates. They are different in that CD-A* can enumerate the candidates in best-first order, while DAA can guarantee that all candidate relaxations generated are minimal. By combining their capabilities, BCDR is able to satisfy all the objectives we proposed in Introduction:

- Simple interaction: we generate only minimal relaxations to inconsistent temporal problems; Enumerate minimal relaxation sets in best-first order according to a preference model. Present the cause of failure (minimal conflicts) to the user.

- Quick response: We use conflicts to guide the enumeration of temporal relaxations and prune search space. The minimal relaxations are generated incrementally.

The program flow of BCDR is shown in (Figure 3-3). BCDR enables the incremental generation of relaxations to OCSTN. BCDR can be terminated at any time, and the first K relaxations generated are the K best ones. It adapts the conflict-directed search techniques used in CD-A* and DAA to guide the search over candidate space.

The pseudo code of BCDR is given in (Algorithm 1). Similar to CD-A*, BCDR starts with the generation of the best candidate minimal temporal relaxation, *currCand* (Step Select Candidate), which suspends no constraints and makes the optimal assignments to each decision variable (Function BestCandidate, Line 1). In Line 7, the function ConsistencyCheck tests *currCand* for consistency (Step Check Consistency). If *currCand* is consistent, it will be added to the results

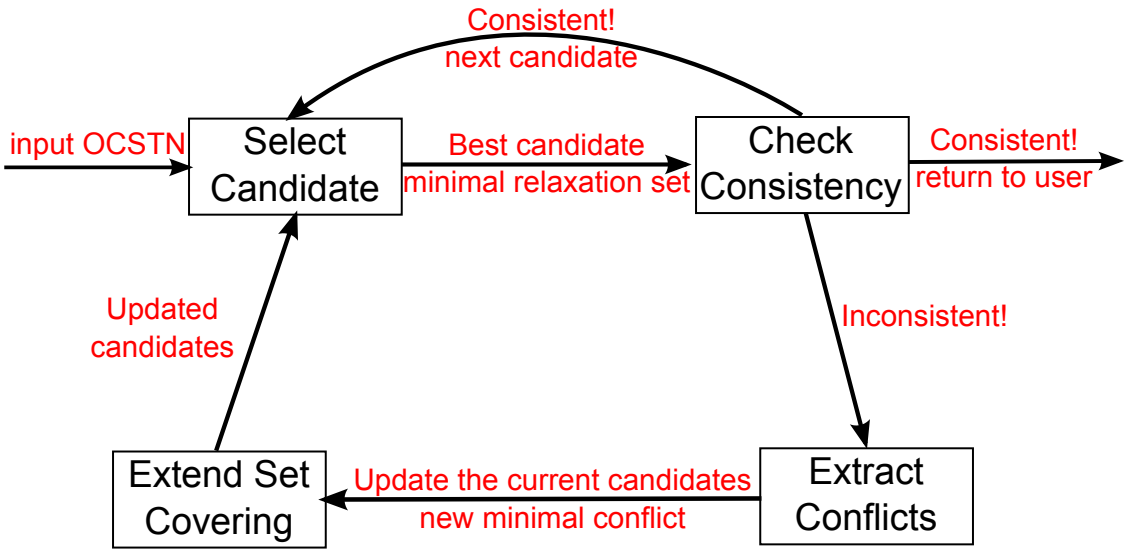


Figure 3-3: The program flow of Conflict-directed Relaxation Enumeration

DMRs (Discrete Minimal Relaxations, Line 8). Then in Line 6, the function DequeueBestCandidate (Step Select Candidate) tests the next most preferred candidate from *CANDs*.

If *currCand* is inconsistent, BCDR will extract a minimal conflict (Function ExtractMinConflict, Line 11, Step Extract Conflicts) and update all existing candidates using the newly discovered conflict (Function UpdateCandidates, Line 12, Step Extend Set Covering). This function implements an incremental hitting set algorithm to generation **minimal** candidate relaxations. We also developed a new inference based conflict extraction algorithm for temporal problems that works one order of magnitude faster than the standard technique used in DAA. Finally, the minimal relaxations generated by BCDR will be mapped back to the temporal constraints in the TPN before being presented to the user.

input : *OCSTN*, an inconsistent OCSTN
input : *UPM*, the user preference model associated with the temporal constraints and decision events in *OCSTN*
input : *K*, number of minimal relaxations required by the user
output: *DMRs*, *K* discrete minimal relaxations to *OCSTN* or all available discrete minimal relaxations, whichever is larger.

```
// Initialize.
```

1 $CANDs \leftarrow \{$BestCandidate($UPM, OCSTN$)$\}$: Assign the best domain value to each decision events without any relaxation.;
2 $DMRs \leftarrow \{\}$: No results generated yet;
3 $CFLTs \leftarrow \{\}$: No conflicts found yet;
4 $i = 0$: Reset result counter;

```
// Generate new candidate and test until the maximum number is
   reached, or run out of candidates.
```

5 **while** $i < K$ **do**
6 $currCand \leftarrow$ DequeueBestCandidate($CANDs, OCSTN, UPM$);
```
   // If consistent, record the current candidate; Otherwise
      extract new conflict and update existing candidates
```
7 **if** ConsistencyCheck*(currCand)* **then**
8 $DMRs \leftarrow DMRs \cup currCand$;
9 $i \leftarrow i + 1$;
10 **else**
11 $CFLTs \leftarrow CFLTs \cup$ ExtractMinConflict($currCand$);
```
      // Generate new minimal relaxation candidates.
```
12 $CANDs \leftarrow$ UpdateCandidates($CFLTs, CANDs$);
13 **end**
```
   // If all candidates have been checked and are consistent,
      no more minimal relaxations can be generated.  Return the
      DMRs.
```
14 **if** $CANDs = DMRs$ **then**
15 **return** *DMRs*;
16 **end**
17 **end**
18 **return** *DMRs*;

Algorithm 1: Main Algorithm of BCDR: the discrete relaxation version

3.4 Related Work

In this section, we present two conflict-directed algorithms that were developed to resolve inconsistent discrete domain constraint satisfaction problems: Conflict-directed A* [37] and Dualize & Advance [4]. We start with a brief review of the algorithms, then discuss their limitations when being applied to inconsistent conditional temporal problems. Finally, we present the ideas that are leveraged off by our BCDR algorithm in resolving over-constrained OCSTNs.

3.4.1 Conflict-directed A*

Conflict-directed A* was developed to solve Optimal CSPs. It can be applied to generate the likely diagnoses to a faulty system if we view diagnosis as a form of constraint suspension [9, 10]. CD-A* uses a best-first search strategy while making use of the conflicts to prune the search space and guide the search. Compared to a constraint-based best-first enumeration algorithm (constraint-based A* and [14, 18, 17]), the conflict-directed technique speeds up the process by almost one order of magnitude [37]. Here we demonstrate CD-A* through a diagnosis problem, one important application of this algorithm. It uses the failure likelihood of components, represented by metric probabilities, to enumerate likely diagnoses in best-first order.

For example, (Figure 3-4) shows a set of cascaded inverters: A, B, C and D. They have different rates of failure, from 1% to 4%. Both the input and output of the system are one. One measurement in the middle of the system indicates zero. If one inverter is working properly, the output of it will be the negation of the input. But, if one inverter is broken, the output can be either positive or negative one, regardless of its input.

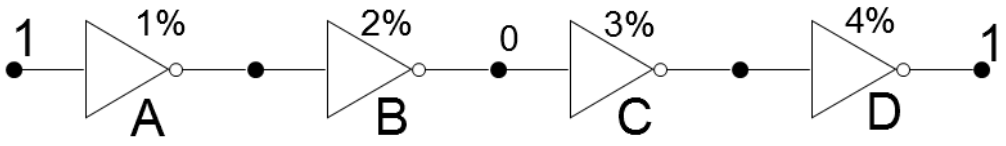


Figure 3-4: Cascaded inverters with different rates of failure

Diagnosis proceeds by making mode assignments, guessing and checking the behavior mode of each component. In this simple example an inverter can be GOOD or BROKEN. Here, making a mode assignment to one component can be viewed as imposing a constraint between the input and output of the component. Due to the identical input and output value, there must be something broken in the system. To find the most likely diagnosis, CD-A* starts with the candidate of the highest probability: A,B,C,D = Good. It has a probability of 90.3%, but is inconsistent with the negative measurement in the middle of the system. Given this inconsistency, CD-A* may extract a conflict from this mode assignment, A,B = Good.

Next, to resolve this conflict, at least one component in A and B has to be broken. CD-A* chooses the one with the higher likelihood, B, and generates a full assignment from it: A,C,D = Good and B = Broken. This assignment is still inconsistent, since C,D=Good is inconsistent with the measurements at the middle and the end.

Finally, CD-A* generates an assignment that can resolve both known conflicts at the same time: A,C = Good and B,D = Broken. It chooses D to resolve the conflict, since D's rate of failure is higher than C. This assignment, is consistent with all the measurements and observations, and is the most likely diagnosis among all consistent diagnoses.

However, using CD-A* to enumerate relaxations raises one important issue: CD-A* would enumerate full relaxations instead of minimal relaxations, hence the number of results generated by CD-A* is usually large and makes it hard to effectively communicate with the users. For example, considering the simple diagnosis problem we presented in (Figure 3-4), nine different full diagnoses can be generated. On the other hand, only four minimal diagnoses exist for the problem. Therefore, we are only interested in generating minimal relaxations. For two relaxations $\mathcal{A}$ and $\mathcal{B}$, we will always prefer $\mathcal{B}$ over $\mathcal{A}$ if $\mathcal{B}$ is a proper subset of $\mathcal{A}$, that is, $\mathcal{B}$ is non-minimal. Hence we never present non-minimal relaxations to the users. However, non-minimal relaxations, like the supersets of existing relaxations, may be generated by CD-A* due to its use of tree expansions: if some assignments are redundant but are located at the root of the search tree, CD-A* won't be able to remove them from the results.

The key idea we leveraged from CD-A* is the conflict-directed search strategy. Consistency-based diagnosis using constraint suspension [9] is a form of relaxation problems with discrete domains: inconsistent mode assignments are the conflicts and diagnoses are discrete relaxations to the problems. CD-A* avoids all known conflicts while generating relaxation candidates. CD-A* starts with the best candidate and tests consistency. If inconsistent, a conflict will be extracted and used to split the tree expansion: the new candidate generated must resolve this conflict by flipping at least one assignment to the variables of the broken system.

3.4.2 Dualize & Advance

The Dualize & Advance algorithm generates discrete minimal relaxations to inconsistent constraint satisfaction problems. Given an inconsistent problem, Dualize & Advance extracts a conflict by iterating through all constraints. It explores the duality between minimal conflicts and minimal relaxations, that is, minimal relaxations are the hitting sets of minimal conflicts, and vice versa. Similar concepts has been implemented in the *General Diagnosis Engine (GDE)* and *Sherlock* [10, 11], a constraint relaxation algorithm which uses the hitting sets of known conflicts between mode assignments to generate candidate diagnoses. However, neither GDE nor Sherlock guarantees the minimality of the relaxation. Using this principle, DAA computes both minimal conflicts and relaxations incrementally: known minimal conflicts are used to computer candidate relaxations through hitting sets, while candidate relaxations are used to discover unknown conflicts until all conflicts in the problem are revealed.

For example, if we apply DAA on the previous diagnosis problem, it will start by testing candidate A,B,C,D = Good. Assume that we find all two minimal conflicts in the cascaded inverters problem through testing this candidate (Figure 3-4): A,B = Good and C,D = Good. DAA can then generate all minimal relaxations by computing the hitting sets of the conflicts (Figure 3-5):

However, DAA is insufficient to solve relaxation problems due to the following two reasons:

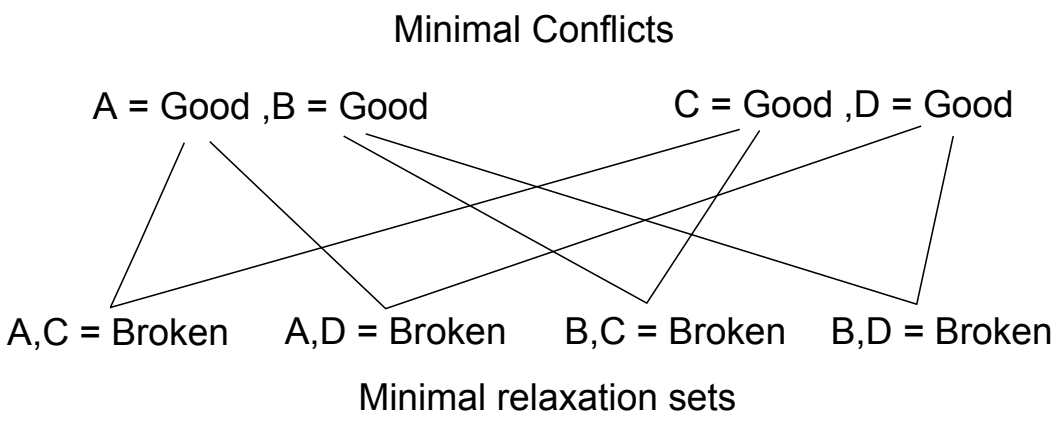


Figure 3-5: Duality between minimal conflicts and minimal relaxation sets

- First, DAA does not consider the users' preferences. Unlike CD-A*, it cannot generate results in best-first order.
- Second, DAA cannot solve conditional CSPs. It requires a conjunctive set of constraints.

BCDR leverages two ideas from DAA in solving relaxation problems. First, DAA takes an incremental approach to detect conflicts and generate valid relaxations. For example, given a set of known conflicts, $CFLTs$, It computes a set of candidate relaxations, $CANDs$, using the hitting sets of $CFLTs$. If one of the candidates in $CANDs$, $cand_i$, is still inconsistent, it implies that at least one conflict has not been detected yet. DAA then extracts one minimal conflict from $cand_i$, update all candidates in $CANDs$ through the incremental hitting set algorithm, and check their consistency.

Second, DAA terminates its enumeration if all candidates in $CANDs$ are consistent, which signals that no more minimal conflicts can be found and all minimal relaxations have been discovered.

3.5 Generating Candidate Relaxations from Conflicts

First, we present the core procedure of BCDR, the *incremental* generation of candidate relaxations. This procedure is implemented as function UpdateCandidates, which computes all consistent candidates based on known conflicts in an OCSTN. In this section, we start with the generation of relaxations to one single minimal conflict, then describe our expansion that resolves multiple conflicts incrementally.

3.5.1 Generating the Constituent Relaxation of a Conflict

To resolve a conflict, one or more temporal constraints in the conflict need to be relaxed. Recall that if any constraint is removed from a **minimal conflict**, the minimal conflict will be resolved. This is the key property of minimal conflicts, and is used by BCDR to resolve conflicts detected in inconsistent temporal problems.

For example, in (Figure 3-6), constraints 'Drive to Quiznos', 'Have dinner', 'Drive home' and 'Time Constraint' form a minimal conflict. To resolve the conflict, we can suspend any one of its constraints, such as the temporal constraint that specifies the overall duration, 'Time Constraint [0min,60min]' (Figure 3-6).

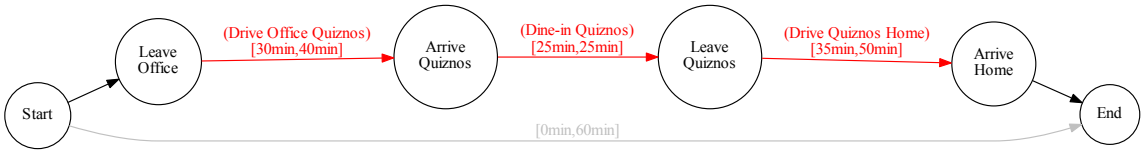


Figure 3-6: Resolving a minimal conflict by suspending one temporal constraint

In addition, for a minimal conflict detected in an OCSTN, one may resolve it by changing the decision that supports the constraints involved in the conflict. Recall that if the label of an OCSTN constraint is not satisfied, the constraint will not be active. It is equivalent to suspending constraints and can resolve the conflict, too. For example, the previously mentioned minimal conflicts require the following decisions:

'LeaveOffice:Drive Office Quiznos', 'ArriveQuiznos: dine-in'.

Modifying either decision would resolve the minimal conflicts, since some of the labeled temporal constraints in the conflict are no longer active. BCDR uses both of these two methods to resolve minimal conflicts detected during the enumeration process, and generates constituent relaxation candidates to inconsistent OCSTNs.

3.5.2 Generating the Constituent Relaxations of Multiple Conflicts Incrementally

Next, we describe the approach that resolves multiple conflicts. As stated in Chapter 2, a valid relaxation must resolve all conflicts in an inconsistent temporal problem. Given multiple minimal conflicts, the relaxation must suspend at least one constraint in each conflict, which makes it a covering set of all minimal conflicts. Therefore, a discrete minimal relaxation can be defined as the minimal covering set of all minimal conflicts in an inconsistent problem [10, 4].

However, it is difficult to accurately identify all minimal conflicts in a problem prior to the enumeration. Detecting all the minimal conflicts requires a lot of computation: for example, in the case of an OCSTN, the algorithm goes through each negative cycle in each component STN. A better approach is to use an incremental search strategy that constructs candidate relaxations based on known conflicts, and updates the candidates when new conflicts are detected. This procedure includes the following steps:

- Generate candidate temporal relaxations based on known conflicts.
- If the candidate is consistent, return as a valid minimal relaxation and move to the next candidate. This is different from CD-A*, which continues extending the search tree. In addition, this minimal relaxation is added to the collection of known conflicts so that the same relaxation will not appear again.
- If the candidate is inconsistent, extract a new minimal conflict from it and

update existing candidates using the conflict (Function UpdateCandidates). Through the update procedure, we can guarantee that all candidates can resolve at least known conflicts.

- If all candidates are tested to be consistent, terminate the enumeration. It indicates that no more conflicts can be discovered, and hence all minimal relaxations have been found.

This is very similar to an incremental repair procedure: a candidate is repeatedly improved by newly discovered conflicts until it makes the over-constrained problem consistent. CD-A* focuses minimal covering by performing expansion in best first order. In BCDR, an incremental set covering algorithm is used to generate candidate minimal relaxation sets using sequentially discovered minimal conflicts (Algorithm 2).

input : *NewMinCFLT*, newly discovered minimal conflict
input : *PrevCandidates*, previous candidate minimal relaxation sets
output: *UpdatedCandidates*, updated candidate minimal relaxation sets

```
// Generate constituent relaxations of the minimal conflict
```
1 *ConstituentRelaxations* ← ResolveConflict(*NewMinCFLT*);
```
// Start with the cross product of previous candidates and the
   new conflict.
```
2 *UpdatedCandidates* ← *PrevCandidates* ⊗ *ConstituentRelaxations*;
```
// Remove redundant (non-minimal) candidates.
```
3 *UpdatedCandidates* ← RemoveRedundantCandidates(*UpdatedCandidates*);
4 **return** *UpdatedCandidates*

Algorithm 2: UpdateCandidates

For example, assume that BCDR tests the first candidate and gets a minimal conflict (highlighted arcs in Figure 3-7).

This minimal conflict contains four temporal constraints, 'Drive Office Cosi', 'dine-in Cosi', 'Drive Cosi Home', 'TimeConstraint', and two decisions, 'LeaveOffice:Drive Office Cosi', 'ArriveDD:dine-in Cosi'. To resolve it, we can suspend any one temporal

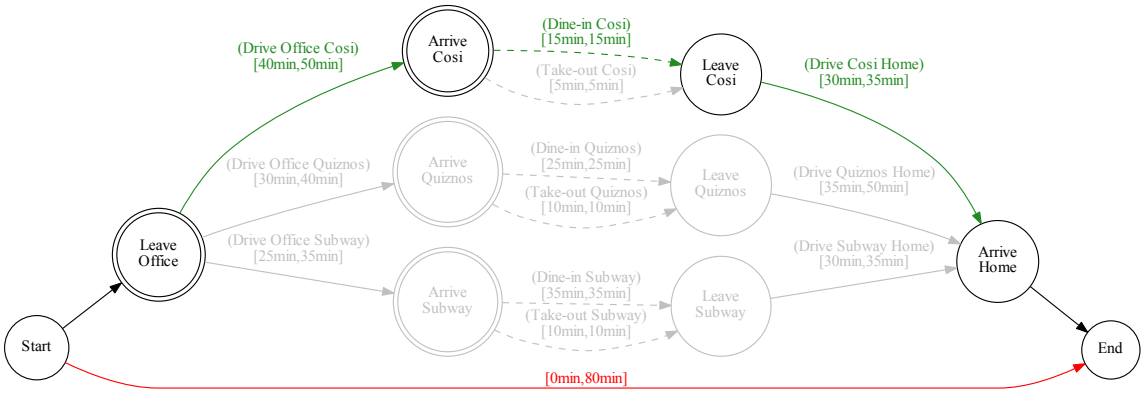


Figure 3-7: A minimal conflict in John's trip plan

constraint or switch one of the decisions. In total, five candidate minimal relaxations can be generated:

Suspend: 'dine-in Cosi' or 'TimeConstraint'

Switch choice: 'ArriveDD:take-out Cosi' or 'LeaveOffice:Drive Office Quiznos' or 'LeaveOffice:Drive Office Subway'.

Next, assuming that BCDR takes the last candidate, it will expand the candidate to 'LeaveOffice:Drive Office Subway ArriveSubway:Take-out Subway'. The consistency check will then indicate that the candidate is still inconsistent and a new minimal conflict can be extracted (Figure 3-8).

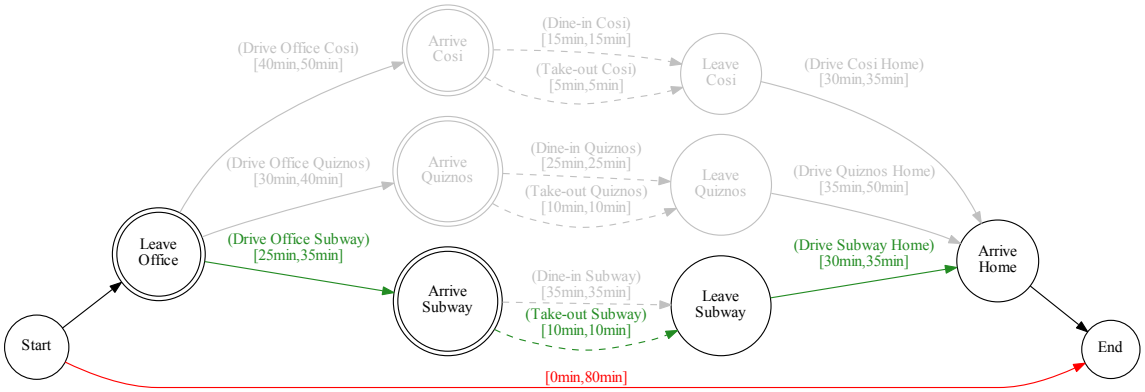


Figure 3-8: Another minimal conflict in John's trip plan

With the discovery of this new minimal conflict, BCDR then updates the current list of candidate temporal relaxations by removing ineffective candidates and adding

new ones. It computes the minimal covering sets of all minimal conflicts found so far, making sure that each candidate resolves all of them. If one candidate has conflicting decisions, for example, 'LeaveOffice:Drive Office Subway' and 'LeaveOffice:Drive Office Quiznos', the candidate will be removed from the list, too.

Suspend: 'TimeConstraint' or 'dine-in Subway' or 'dine-in Cosi'

Assign: 'LeaveOffice:Drive Office Quiznos' or 'ArriveSubway:take-out Subway' or'ArriveCosi:take-out Cosi'

Note that this process is different from iterative repair algorithms, which are stochastic in that they cannot guarantee the completeness and optimality of the results. Function UpdateCandidates guarantees that BCDR finds all minimal conflicts and corresponding minimal temporal relaxations to an inconsistent OCSTN, given enough iterations. It terminates when all the candidates in *CANDs* are consistent. It indicates that all minimal conflicts in the inconsistent problem have been detected, and no more minimal relaxation can be generated.

Finally, we prove the completeness and soundness of BCDR in resolving over-constrained OCSTNs.

Theorem 3. *[Completeness of BCDR] BCDR can find all minimal relaxations given an over-constrained OCSTN.*

Proof. [Proof by contradiction]

Let MR be a minimal relaxation to an OCSTN $\mathcal{P}$, and BCDR fails to generate it.

- Since the incremental set covering method used by BCDR is complete, if MR is not generated by BCDR, at least one minimal conflict, $MinCFLT$, has not been detected by BCDR yet. Otherwise all minimal relaxation sets to $\mathcal{P}$ must have been generated.
- If $MinCFLT$ is unknown to BCDR, some of the candidate relaxations generated by BCDR must be inconsistent when BCDR terminates, since not all of the candidate relaxations can resolve the unknown $MinCFLT$.

- However, this contradicts with the termination condition of BCDR: BCDR will only terminate if all candidates are minimal relaxations that resolve all conflicts.

  Hence the assumption is faulty. BCDR is complete. □

Theorem 4. *[Soundness of BCDR] All the results generated by BCDR are minimal relaxations that resolve all conflicts in an OCSTN.*

Proof. A minimal relaxation generated by BCDR is correct in two aspects:

1. Must be a valid relaxation: since all relaxations generated by BCDR must pass consistency check before being returned to the user, they have to resolve all the conflicts in the over-constrained OCSTN.

2. Must be minimal: this is guaranteed by the minimal set covering process. All candidates generated are the minimal covering sets of known conflicts.

Hence all minimal relaxations returned by BCDR are correct, that is, are *minimal* and resolve *all* conflicts. □

In summary, we presented the candidate generation method used in BCDR. Given an inconsistent OCSTN, the method is guaranteed to find all discrete minimal relaxations that resolve all its conflicts. We make use of the property of minimal conflicts, for which a conflict can be resolved by suspending any one constraint in it. We then compute candidate relaxations through a minimal set covering process. In addition, to improve the performance in real world applications, we developed the incremental approach UpdateCandidates that generates candidate based on known conflicts, then makes updates when new conflicts are detected. Therefore, the first minimal relaxation will be generated as early as possible, and even prior to the discovery of all conflicts.

3.6 Selecting Preferred Candidates

The problem we are addressing in this section is: given a set of candidate relaxations and a user preference model, select the most preferred one from the set. We present the candidate selection procedure of BCDR, GenerateCandidate, which evaluates the candidates and selects the most preferred one from all candidates generated in Section 3.4. It guides the enumeration towards the most preferred relaxation and guarantees that BCDR enumerates minimal relaxations in best-first order, according to predefined user preference models.

The users usually have preferences over temporal constraints in the problems. For example, John may have a very important party at home, thus he would rather shorten his dinner time to delay this arrival. Considering user preferences during the relaxation process can significantly improve the efficiency while communicating the temporal relaxations to the user: if resolutions are generated in best-first order according to the user's preferences, there will be a much higher chance that the passenger can find his preferred relaxation without going through too many iterations with Uhura. This is one of the key enablers of simple interaction in collaborative plan diagnoses.

In Section 3.2, we defined the preference models that could be used in relaxation problems. Here we present one such model that satisfies the requirements: a metric cost function over constraints and decision events. Section 3.6.1 describes the outer loop function, GenerateCandidate, and in Section 3.6.2 we present a quantitative preference model that can be integrated with BCDR.

3.6.1 Selecting the Most Preferred Candidate

The key to improving user interactions is to find the preferred resolution as quickly as possible. Therefore, BCDR enumerates temporal relaxations in best-first order to increase the opportunity that the user agrees to one of the first several resolutions. Given a list of candidate minimal relaxations and a user preference model, Function DequeueBestCandidate finds the most preferred candidate in that list. This

problem is formally defined as the following:

Definition 35. *(Find the Most Preferred Candidate) Given a list of candidate minimal temporal relaxations CANDs and a preference model UPM, return the candidate that is most preferred by the user compared to all the other candidates in the list according to UPM.*

- *The user preference of a minimal relaxation, MR, is evaluated based on the best relaxation covered by MR. In other words, the user preference over MR is the preference of the best temporal relaxations among its supersets.*
- *The preference model, UPM, can be used to compare two candidate temporal relaxations and return the preferable one. If two candidates are equally preferable or not comparable, UPM returns both candidates or signal failure.*

Any preference models that can be used to compare two relaxations can be used in BCDR. Given such a preference model, DequeueBestCandidate is guaranteed to find the most preferred candidate in a list of candidates. In this section, we demonstrate BCDR using a metric cost function over the temporal constraints and decision events in an OCSTN.

Function DequeueBestCandidate (Algorithm 3) selects the best candidate through a series of binary comparisons (Function Better?, Line 3). It takes an A* like approach: instead of comparing the partial choices and relaxations generated by the minimal set covering process, it expands both candidates to the best full candidates that subsume them before the comparison. This guarantees that DequeueBestCandidate generates the candidate that leads to the best minimal relaxation to the problem.

The additional expansion step is implemented in (Function Expand of Algorithm 3). Due to the inherent property of the minimal set covering procedure (Algorithm 2), the candidates generated from conflicts are usually incomplete relaxations, such as selecting 'LeaveOffice:(Drive Office Quiznos)' and suspending 'Time Constraint'. There might be unassigned decision variables in the candidate.

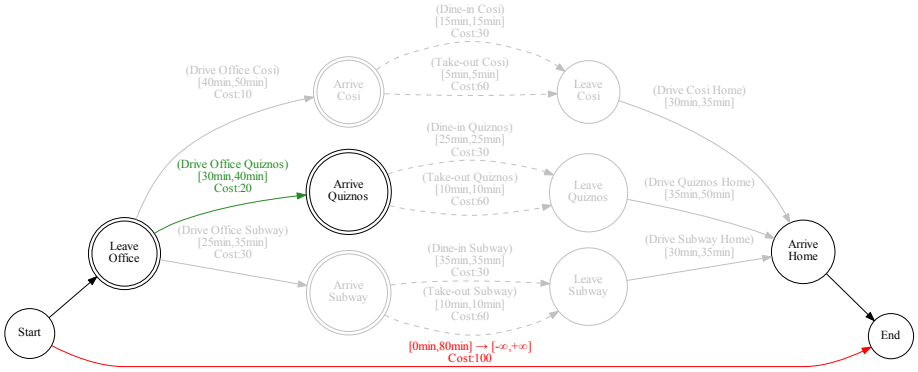
```
input  : CANDs, a list of candidate discrete minimal relaxation sets
input  : TPN, the over-constrained TPN
input  : UPM, the user preference model
output: BestCandidate, the best candidate in CANDs

// Initialize BestCandidate with the first candidate in the
   list.
1 BestCandidate ← GetFirst(CANDs);
// Loop through the list of candidates; select the best one
   through a series of binary comparisons.
2 for currCandidate in CANDs do
3 |   if Better?(Expand(currCandidate),Expand(BestCandidate)) then
4 |   |   BestCandidate ← currCandidate
5 |   end
6 end
7 return BestCandidate
```

Algorithm 3: DequeueBestCandidate

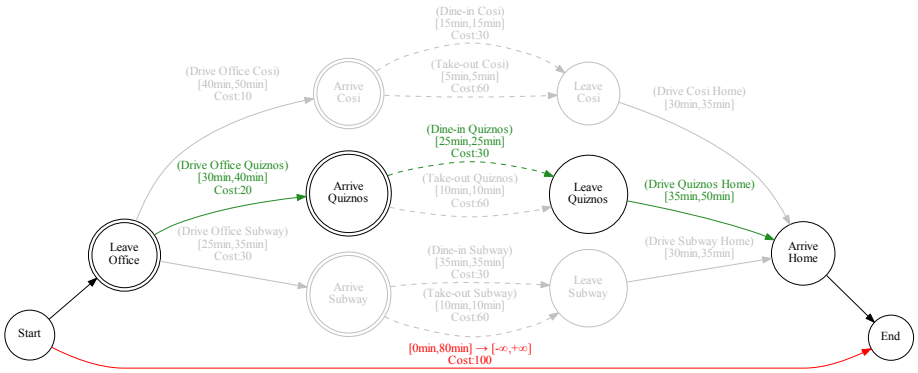
Therefore, BCDR uses a standard A* approach: instead of comparing the costs of partial candidate, it compares the best complete candidate that subsumes the partial candidate. This is similar to the admissible heuristics that provides a bound on the cost of extending this partial candidate. The expansion procedure involves two steps:

- First, BCDR expands a partial candidate by adding all necessary decisions to activate the constraints in it.
- If there are decision variables in the OCSTN left unassigned after the first step, BCDR will make choices to them until the candidate is complete. In the second step, all choices made by BCDR select the best option within the domain of decision events.

For example, a partial candidate from the previous example is 'LeaveOffice:(Drive Home Quiznos)' and suspending 'Time Constraint' (Figure 3-9(a)). The decision at 'ArriveQuiznos' needs to be made. The expansion step selects the domain value with the lowest cost for this unassigned decision event, '(dine-in Quiznos)', and avoids suspending any more constraints to minimize the cost. (Figure 3-9(b)).



(a) Incomplete Candidate



(b) Expanded candidate

Figure 3-9: Examples of expanding incomplete candidates

We now prove the optimality of BCDR.

Theorem 5. *Given an inconsistent OCSTN, $\mathcal{P}$, and a valid preference model, UPM, the first minimal relaxation returned by BCDR is the most preferred one according to UPM, given that Function* DequeueBestCandidate *always returns the best candidate.*

Proof. [Proof by contradiction]

Assume that the minimal relaxation generated by BCDR, *MR*, is not the best one. There is another minimal relaxation, *MR*′, which is more preferred by the user.

Both *MR* and *MR*′ are valid relaxations. Therefore, they resolve all the conflicts, *allCFLTs*, in $\mathcal{P}$.

Next, *MR* must be generated from a set of minimal conflicts, *knownCFLTs*, which is a subset of *allCFLTs*. Given that DequeueBestCandidate returns the most

preferred candidate in the list, MR is the best relaxation that can resolve all the conflicts in *knownCFLTs*.

However, MR' also resolves *knownCFLTs*, since it is a valid relaxation and must resolve all conflicts.

Therefore, MR' cannot be better than MR. The assumption does not hold.

BCDR generates the best minimal relaxation given an over-constrained OCSTN.

□

Finally, the preference model is implemented in the BETTER? function, and must satisfy the requirement specified in this section. The model is only used to compare two complete candidate relaxations, and it is the only preference model specific part of Function DEQUEUEBESTCANDIDATE.

3.6.2 Modeling Preference using Metric Costs

We now demonstrate a preference model in temporal relaxation problems, the metric cost function. It models the users' preference with quantitative values: each constraint is given a cost value and the planner is designed to find the plan with the lowest cost [28]. The metric cost function is useful in that it accurately captures the relative preference over different relaxations, and is easy to compute.

In temporal relaxation problems, the goal is to satisfy as many user preferred constraints as possible [30]. The metric cost model associates each decisions and constraints with real numbers: the cost of a decision is *received* if one relaxation makes the choice; and the cost of a constraint is received if the temporal constraint is suspended in a relaxation. This is similar to a weighted constraint in MaxSAT problems [2].

Definition 36. *(Simple Metric Cost Functions) A metric cost function of an OCSTN, $\mathcal{F}$, is a mapping from constraints and decisions in the OCSTN to real values where:*

- *Each temporal constraint, tc_i, is mapped to a real value $c_{tci} \leftarrow \mathcal{F}(tc_i)$ representing the cost if this temporal constraint is suspended. The cost is zero if tc_i is preserved.*

- *Each label, de_j, of an each decision variable, DE_i, is mapped to a real value $c_{dej} \leftarrow \mathcal{F}(de_j)$ representing the cost of choosing de_j.*

- *The total cost of a temporal relaxation is defined as the sum of the costs of decisions and all suspended temporal constraints: $Cost \leftarrow \sum_1^m \mathcal{F}(tc_i) + \sum_1^n \mathcal{F}(de_j)$.*

- *Given two temporal relaxations, $\mathcal{TC}_a$ and $\mathcal{TC}_b$, the users prefer $\mathcal{TC}_a$ if $\textsc{Cost}(\mathcal{TC}_a) < \textsc{Cost}(\mathcal{TC}_b)$, and vice versa.*

The term 'received' implies that the cost is incorporated into the overall cost of the solution. In this subsection, we demonstrate the enumeration process using an additive objective function. For example, in (Figure 3-10), decision variable 'Leave-Office' has three labels with different costs: '(Drive Office Cosi)'(10),'(Drive Office Quiznos)'(20) and '(Drive Office Subway)'(30). The label with the lowest cost, 'Leave-Office:(Drive Office Cosi)' is the most preferred one. The same principle applies to constraints: suspending '(dine-in Cosi)'(30) is preferred to suspending 'TimeConstraint' (100).

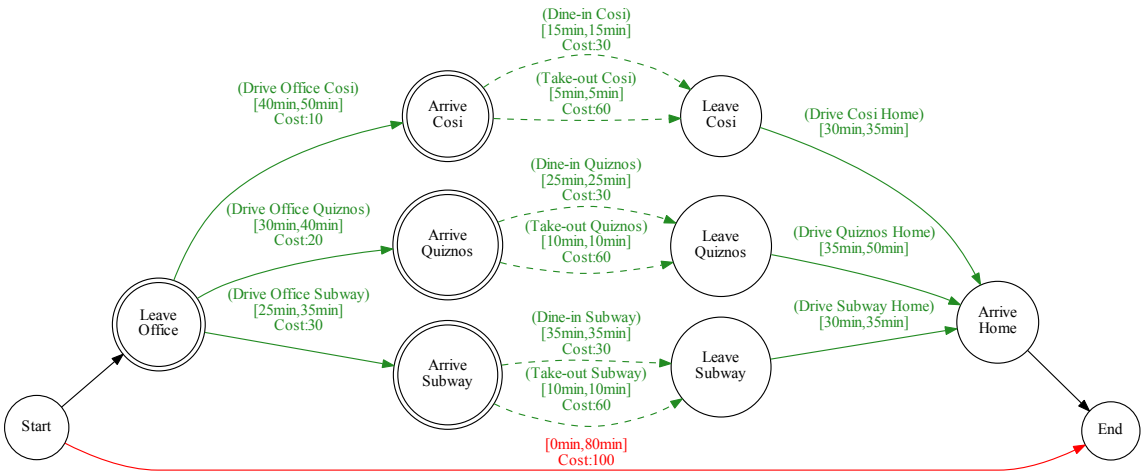


Figure 3-10: A real valued objective function for John's trip plan TPN

The cost of a minimal relaxation is computed by summing up the costs of all decisions and suspended temporal constraints in it. For example, (Figure 3-11) shows a relaxation in which John will go and have dinner at Cosi without satisfying the overall time constraint. The cost of this solution is:

F(Drive Office Cosi) + F(dine-in Cosi) + F(suspend 'TimeConstraint')

$10 + 30 + 100 = 140$

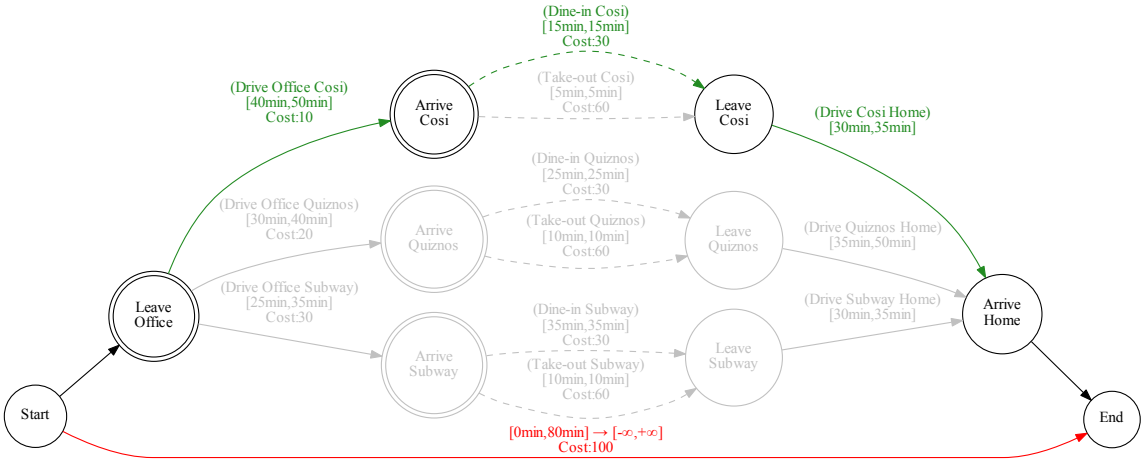


Figure 3-11: The real valued cost for a minimal relaxation set

To compare two candidates with metric cost, one may simply compare the sum of their costs of choices and constraint suspension. For example, the cost of the candidate in (Figure 3-9(b)) is 150, which is larger than the cost of (Figure 3-11). Hence, F(Drive Home Cosi) + F(Dine-in Cosi) + F(suspend Time constraint) is a more preferred resolution to F(Drive Home Quiznos) + F(Dine-in Quiznos) + F(suspend Time constraint).

In summary, we presented the method that selects the best candidate. Given a list of candidate relaxations, the Function DequeueBestCandidate is able to return the best one according to a preference model. Using this method, we proved that BCDR generates minimal relaxations for over-constrained temporal problems in best-first order.

3.7 OCSTN Consistency and Conflict Detection

In this section, we present the method that checks if an OCSTN is consistent and detects the cause of failure (conflicts) if the OCSTN is inconsistent. We first give a brief review of the negative cycle detection algorithm, a standard method for consistency checking of temporal constraint networks. Then we present a domain-specific inference-based method, ExtractMinConflict, which extracts minimal conflicts from inconsistent OCSTNs. It explores the connection between conflicts and negative cycles in a temporal constraint network, and performs one order of magnitude faster than the methods used in ([4, 29]) for STNs.

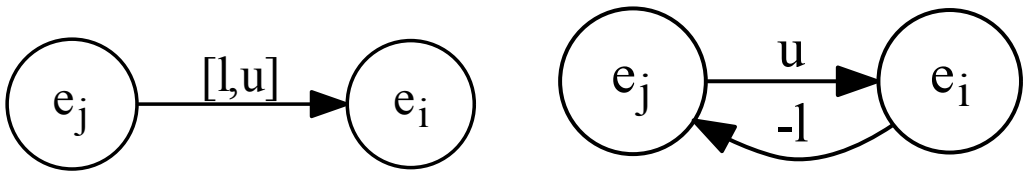
BCDR implements the Incremental Temporal Consistency algorithm [20] and Bellman-Ford algorithm [7] in the ConsistencyCheck function to check the consistency of Simple Temporal Networks. Both algorithms look for negative cycles in the equivalent distance graphs of STNs to determine consistency. If a STN is inconsistent, there must be negative cycles in the network, which indicates the existence of conflicts between the constraints. ExtractMinConflict then maps the negative cycles to conflicting sets of constraints and reveal the core cause of inconsistency.

3.7.1 Consistency Checking as Negative Cycle Detection

We start with a review of consistency checking of simple temporal networks. A STN can be viewed as a special case of a *linear programing* problem, in which linear constraints are replaced by simple temporal constraints. One may use general LP algorithms, such as Simplex algorithm [8] and interior-point method [21], to find solutions to simple temporal networks. However, these algorithms are designed to find the optimal solutions of LP problems instead of checking consistency, hence not efficient if we only want to know about the consistency of a temporal network. In fact, it has been shown that due to the special formulation of STNs, an equivalent distance graph always exists for any STN and the consistency can be determined by negative cycle detection algorithms, which are significantly more efficient.

A STN can be converted to an equivalent **directed constraint graph**. The

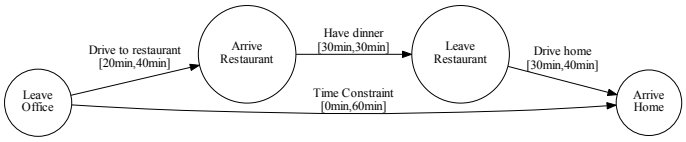
equivalent distance graph, DG, of a STN has the same set of variables. The directed-arcs in DG are generated from the simple temporal constraints in the STN. Each constraint c_i is converted into two directed arcs: one marked with upper bound pointing to the end variable, and the other one marked with the negation of lower bound and pointing to the start variable (Figure 3-12).



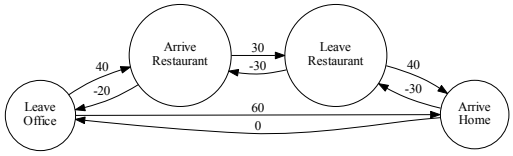
(a) A simple temporal constraint (b) its equivalent arcs in distance graphs

Figure 3-12: Convert a simple temporal constraint to arcs in a distance graph

Following this rule, we can convert the STN in (Figure 3-13(a)) to its equivalent distance graph and check its consistency as a negative cycle detection problem.



(a) An inconsistent STN of John's trip back home



(b) The corresponding inconsistent temporal graph

A negative cycle is a cycle in a distance graph whose weighted directed arcs sum to a negative value. For example, the highlighted arcs in (Figure 3-13) forms a negative cycle, since the sum of the weights of all arcs is -20.

Negative cycles in distance graphs can be detected by many shortest-path algorithms, like Floyd-Warshall [15] and Bellman Ford [7]. For an equivalent distance graph of a STN, its existence indicates that the STN is inconsistent [12]. For reference, the Floyd-Warshall algorithm is provided in (Algorithm 4).

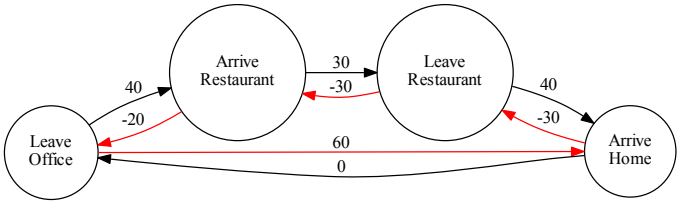


Figure 3-13: A negative cycle in a distance graph

input : V, the set of vertices in a distance graph
output: E, the set of edges in a distance graph

```
1  for i in V do
2  |  for j in V do
3  |  |  for k in V do
4  |  |  |  if E_ij > E_ik + E_kj then
5  |  |  |  |  E_ij ← E_ik + E_kj
6  |  |  |  end
7  |  |  end
8  |  end
9  end
10 for i in V do
11 |  if E_ii < 0 then
12 |  |  return false
13 |  end
14 end
15 return E
```

Algorithm 4: The ConsistencyCheck function using Floyd-Warshall All-Pairs Shortest Path algorithm

For OCSTNs, one can determine if it is temporally consistent by grounding it into component STNs and check the consistency of each component STN using negative cycle detection algorithms. According to the definition (Section 3.1), if one of the component STNs is consistent, so is the OCSTN. Otherwise, the OCSTN is over-constrained, since no guard set can be found that activates a consistent set of constraints. A faster and incremental approach to check the consistency of OCSTNs, *Incremental Temporal Consistency*, is presented in [19, 13].

3.7.2 Extracting Conflicts

Now we present the ExtractMinConflict method that extracts minimal conflicts from inconsistent OCSTNs using inference. It uses an approach that extracts minimal conflicts quickly using the negative cycle detection algorithms, without looping through all constraints in the problem, like the conflict extraction algorithm used by Dualize & Advance. We first describe its implementation on STNs, then present an expansion to ExtractMinConflict that extracts conflicts from OCSTNs.

Conflicts in STNs

First, we define the conflicts of an inconsistent STN:

Definition 37. *A* **conflict** $CFLT$ *of a STN* $< \mathcal{V}, \mathcal{C} >$ *is a subset* $\mathcal{C}' \subseteq \mathcal{C}$ *such that* $\mathcal{C}'$ *is inconsistent. A* **minimal conflict** *of a STN is a conflict* $\mathcal{C}'$ *whose proper subsets are not conflicts.*

In other words, if one constraint is removed from a minimal conflict, $MinCFLT$, then it is no longer a conflict. Given an inconsistent STN, there must be conflict in the STN so that no schedule can satisfy all constraints. For example, in John's trip plan, his time constraint is too small compared to the time required by his trip: the sum of driving and dinner durations is at least 80 minutes, which is much larger than the time constraint (60 minutes).

The set of all constraints of an inconsistent STN forms a conflict. However, in most cases it is more useful to consider conflicts that are minimal: A conflict is an inconsistent set of constraints; A minimal conflict is a conflict such that no subset of it is a conflict. Hence, the removal of any constraint in a minimal conflict restores its temporal consistency. Intuitively, the minimal conflicts can be interpreted as the core causes of failure. In addition,

For example, constraints 'Drive to restaurant', 'Have dinner', 'Drive home' and 'Time constraint' in (Figure 3-13(a)) form a minimal conflict. if John removes his

goal of trip duration, that is, the 'Time Constraint' in the STN, the conflict will be resolved (Figure 3-14).

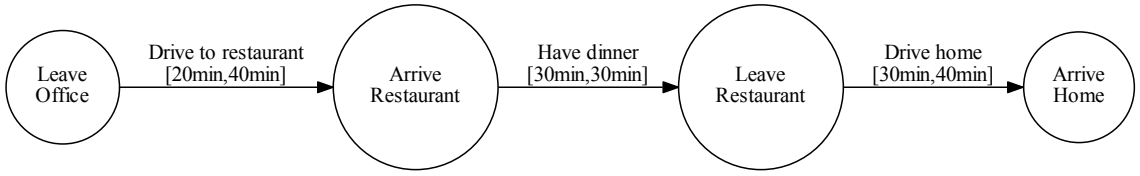


Figure 3-14: John's trip without the temporal goal of duration

Detecting Minimal Conflicts of unconditional STNs

Conflicts can be detected through search, inference or a mix of both. There are multiple ways to detect a minimal conflict in an inconsistent STN. DAA uses a search based approach that iterates through all constraints in a problem and tests which constraint can be removed, while keeping the problem inconsistent. (Function ExtractMinConflict) works in the same way as the GROW function in the Dualize & Advance algorithm (Algorithm 5). This general method works for all constraint satisfaction problems with either discrete or continuous variables. However, it requires K (the number of constraints in the problem) consistency checks regardless of the type of problem. This significantly decreases the efficiency of the relaxation generation process.

There are conflict detection algorithms that largely use inference, and are much faster than algorithms based on search. The most common example is the conflict extraction based on unit propagation and is performed by *Truth Maintenance Systems* [11]. For problems with simple temporal constraints, we developed a faster way to extract the minimal conflict using negative cycles. This is an inference approach that is similar to the conflict extraction method implemented in GDE [10], which detects conflicts through propagating observations through mode assignments.

The existence of a negative cycle indicates that the set of constraints correspond to the cycle is inconsistent. Therefore, if a set of constraints correspond to a negative cycle, they form a conflict. ITC and Bellman Ford algorithms detect negative cycles

input : *currCAND*, an inconsistent candidate relaxation set
output: *minCFLT*, a minimal conflict

`// Initialize.`

1 *minCFLT* ← UnsuspendedConstraints(*currCAND*): Start with the unsuspended constraints in the inconsistent candidate, which is a non-minimal conflict;

`// Loop through all constraints that are active in` *minCFLT*

2 **for** *Constraint in minCFLT* **do**
3 **if** ¬ConsistencyCheck*(minCFLT\Constraint)* **then**
  `// If the removal of` *Constraint* `cannot restore temporal consistency, remove it from the conflict`
4 *minCFLT*←*minCFLT\Constraint*;
5 **end**
6 **end**
7 **return** *minCFLT*;

Algorithm 5: ExtractMinConflict

to determine inconsistency. However, neither ITC nor Bellman-Ford guarantees the minimality of the conflicts extracted. They may return non-simple cycle such as the one shown in Figure 3-15(a). A non-simple cycle is a cycle in a distance graph with repeated vertices, and may correspond to a conflict that is non-minimal. We present the ExtractMinConflict function used by BCDR that extracts minimal conflicts from inconsistent STNs. It adds a post-process to the negative cycles and minimizes them using (Theorem 6).

Theorem 6. *A set of simple temporal constraints that forms one and only one negative cycle without repeating vertex is a minimal conflict.*

Proof. If $\mathcal{T}$ is a set of temporal constraints that contains a negative cycle NC, meaning that $\mathcal{T}$ is a conflict. There is no consistent schedule to the events in $\mathcal{T}$, $\mathcal{EVT}$, that can satisfy all the temporal constraints.

Next, if there is no repeated vertex in NC, one and only one cycle of edges exists in this negative cycle. If one edge is removed, NC will be broken: no negative cycle

in $\mathcal{T}$ exists any more.

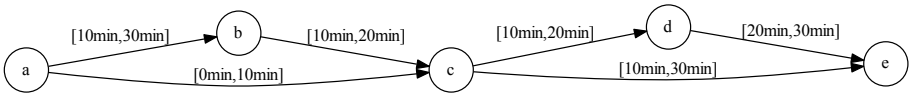
Therefore, for a negative cycle without any repeating vertex, the removal of any temporal constraint will break the negative cycle. Hence the corresponding constraint set of a negative cycle is a minimal conflict. □

BCDR does not need to iterate through all temporal constraints to minimize a conflict. It only needs to look for a set of simple temporal constraints that form a negative cycle where each vertex has exactly one incoming and one outgoing arc (a simple cycle). If there are repeating vertices in a negative cycle, then at least one of the sub-cycles is a minimal conflict. The update function ExtractMinTemporal-Conflict splits the negative cycle at the repeating vertex, checks all the sub-cycles until one without repeating vertex is found.

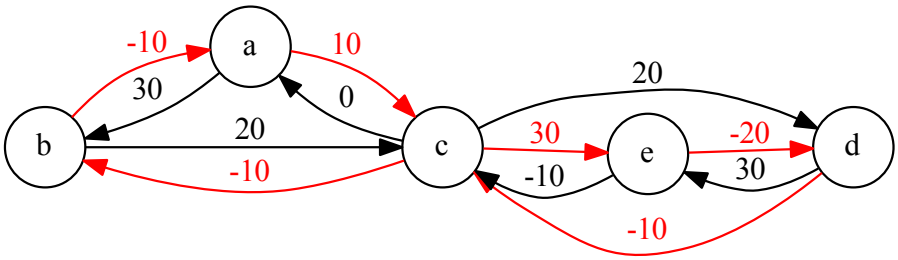
For example, in (Figure 3-15(a)), there are six simple temporal constraints between five variables: a, b, c, d and e. This set of constraints is inconsistent, since a negative cycle can be detected by consistency checking algorithms in its equivalent distance graph (red arcs in Figure 3-15(b)). This cycle goes through all variables and visited c twice. Therefore, this set of all six temporal constraints is not a minimal conflict.

To extract the minimal conflict from this set of temporal constraints, Function ExtractMinTemporalConflict splits the negative cycle at the repeating node, c, and generates two sub-cycles (Figure 3-15(c) and 3-15(d)). Neither of the sub-cycles has repeating node, and only (Figure 3-15(d)) remains a negative cycle. Therefore, the minimal conflict is 'a-b [10min,30min]', 'b-c [10min,20min]' and 'a-c [00min,10min]'.

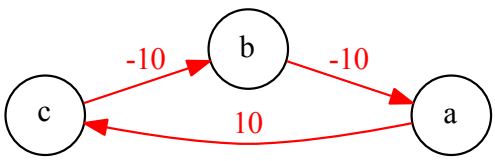
Although the worst case complexity of ExtractMinTemporalConflict (Algorithm 6) is still N (the number of constraints in the conflict), it usually finds the minimal conflict within the first or second iteration in real world scenarios. Because the number of negative cycles in a conflict is usually much less than the number of constraints. It saves many consistency checks and makes the enumeration process of BCDR nearly one order of magnitude faster, in practice.



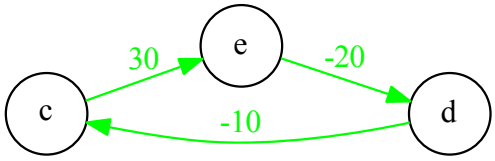
(a) An inconsistent set of constraints



(b) A negative cycle with a repeating node



(c) Inconsistent sub-cycle



(d) Consistent sub-cycle

Figure 3-15: Examples of splitting negative cycles with repeating vertices

Detecting Minimal Conflicts of OCSTNs

We presented the definition of conflicts and minimal conflicts of OCSTNs in Section 3.1. In addition to temporal constraints, the conflicts of OCSTNs also include guards, since its temporal constraints may depend on one or more decisions represented by guards.

For example, if John wants to get home in 60 minutes, (Figure 3-16) becomes a conflict in the conditional STN:

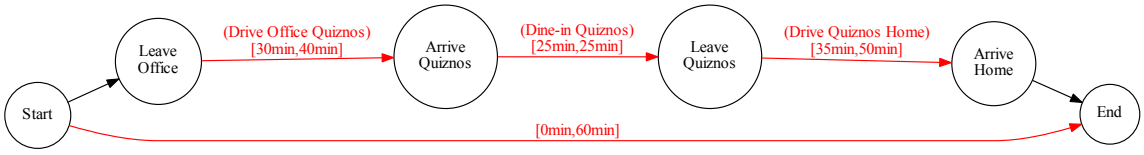


Figure 3-16: An OCSTN with a conflict

```
input  : NCycle, a negative cycle in an inconsistent candidate relaxation set
output: minCFLT, a minimal conflict

// Recursively splitting negative cycles if there is repeating
   vertex.
1  if HasSubCycles(NCycle) then
2  |  NegativeCycles ← SplitNegativeCycle(NCycle);
3  |  for SubCycle in NegativeCycles do
4  |  |  return ExtractMinTemporalConflict(SubCycle);
5  |  end
6  else
7  |  if ¬ConsistencyCheck(NCycle) then
   |  |  // A negative cycle is a minimal conflict if there is no
   |  |     repeating vertex in it.
8  |  |  return NCycle;
9  |  end
10 end
```

Algorithm 6: ExtractMinTemporalConflict using negative cycles

Constraints: 'Drive to Quiznos', 'Have dinner', 'Drive home', 'Time Constraint'

Choices: 'LeaveOffice:Drive Office Quiznos', 'ArriveQuiznos: dine-in'.

'(Drive Office Quiznos)', '(Dine-in Quiznos)', '(Drive Quiznos Home)' and '[0min,60min]' forms an inconsistent set of constraints: the trip takes at least 90 minutes, while the temporal goal is at most 60 minutes. Guard 'LeaveOffice:Drive Office Quiznos' and 'ArriveQuiznos: dine-in' are required to activate these conflicting temporal constraints.

The method of detecting conflicts in OCSTNs is the same as the one presented in Section 3.7.2. Given an inconsistent OCSTN, we first extract the minimal conflicts detected in one of its component STNs. Then we record it with the guards required to activate these constraints in the minimal conflicts. The minimal conflict in an OCSTN is slight different from the minimal conflict in a STN in that it can be resolved by either removing one constraint or changing the decision made to one of the guards. More specifically, if a decision is necessary to enable some of the constraints in the minimal conflict, then changing it will resolve the minimal conflict, since the constraints that are guarded by the decision are no longer activated.

In this section, we presented an innovative approach to detect *minimal conflicts* in temporal problems. *Minimal conflicts* are defined as inconsistent sets of guarded temporal constraints and are the core causes of failure for over-constrained OCSTNs. Our inference-based method extracts minimal conflicts using the negative cycles detected by temporal consistency algorithms. Compared to the search-based conflict extraction method implemented in Dualize & Advance, our method improves the run-time performance by nearly one order of magnitude, enabling BCDR to resolve larger scale problems.

3.8 Chapter Summary

In this chapter we presented the concept and design of *Best-first Conflict-directed Relaxation*, an algorithm that generates preferred discrete minimal relaxations to inconsistent OCSTNs. It is the core method in Uhura that supports the collaborative diagnosis of over-constrained temporal plans. We developed two innovative methods in BCDR that bring it two distinct features compared to previous approaches: quick response and simple interaction.

First, to improve the efficiency in the generation of relaxations, BCDR only enumerates *minimal* relaxations, a compact representation of all relaxations that significantly reduces the size of the search and result space. Previous work can enumerate full relaxations in best-first order or generate all minimal relaxations. BCDR is novel in that it can 1) return the preferred minimal relaxation and 2) the leading N preferred minimal relaxations. BCDR is directly inherited from conflict-directed A* with simple modifications and ideas from Dualize & Advance and other CSP solvers to enumerate relaxations efficiently by (1) using a domain-specific inference-based conflict extract algorithm, (2) guiding the search with minimal conflicts detected in the enumeration and (3) generating relaxations from the hitting sets of minimal conflicts. With the implementation of incremental relaxation generation, BCDR can return relaxations prior to the discovery of all conflicts, which further improves its efficiency in response to the user.

In addition, BCDR generates minimal relaxations in best-first order. It uses a metric cost function over constraints and decisions to prioritize the relaxations. The prioritized results greatly reduces the information exchange between the users and autonomous systems, making the collaborative diagnosis process simpler and more efficient.

BCDR has been implemented in Uhura to support collaborative temporal plan diagnosis through a mapping between TPNs and OCSTNs. We claim that BCDR achieves nearly two orders of magnitude improvements in terms of the run time performance. The benchmark results are presented in Chapter 5.

Chapter 4

Continuous Temporal Relaxations

The discrete relaxation to inconsistent OCSTNs is presented in Chapter 3. This is similar to prior works, which have focused on discrete, rather than partial relaxation of temporal constraints in temporal problems. It simplifies the relaxation process by taking an all-or-nothing approach in which constraints are either preserved or suspended. For example, John realizes that he cannot arrive at the party on time and decides not to go at all. However, a better solution would be to partially relax the constraint by calling his friends and asking for a later starting time. By introducing the concept of continuous relaxation, a weakened version of users' goals can be preserved in the relaxed problem, and no constraint will be suspended but only adjusted. Continuous relaxation addresses the full problem of temporal relaxations. Note that in this example there might be a penalty for being late, which increases with the length of the delay. Similar to generating preferred discrete relaxations, we want to use preference models to generate the most preferred continuous relaxations.

In this chapter we present an innovative method for partial relaxations based on preferences on continuous variables, Continuous BCDR. This addresses the third requirement of collaborative diagnosis: small perturbation. Continuous BCDR resolves inconsistent OCSTNs by relaxing the temporal bounds of constraints. This is a first method that enables the continuous relaxation of temporal problems, in which temporal constraints can be preserved in the relaxations. We present continuous relaxation as a generalization of discrete relaxation, and use the discrete version

of BCDR as a stepping stone to CONTINUOUS BCDR. With a continuous user preference model over temporal constraints, CONTINUOUS BCDR enumerates *minimal continuous temporal relaxations* in best-first order.

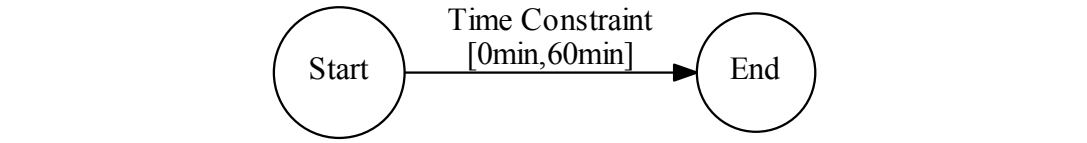
CONTINUOUS BCDR avoids unnecessary relaxations of temporal constraints: it minimally relaxes the temporal bounds only to the degree that is necessary for restoring temporal consistency. Compared to discrete relaxations, continuous relaxations to temporal problems can avoid unnecessary utility loss compared to discrete relaxations. It further improves the quality of the resolutions generated by discrete BCDR, and minimizes the perturbation to the users' goals.

We begin in Section 4.1 by defining the problem of generating continuous relaxations to inconsistent OCSTNs. Then in Section 4.2 we present an overview of CONTINUOUS BCDR, which enumerates continuous temporal relaxations in best-first order. We then describe the new method that generates continuous relaxation candidates from conflicts in Section 4.3. Finally, we present the preference models that can be used with CONTINUOUS BCDR to generated preferred relaxations in Section 4.4.

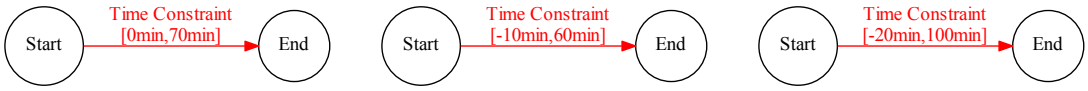
4.1 Problem Statement

In this section, we define the problem of generating continuous relaxations to inconsistent OCSTNs. Similar to discrete relaxation problems, the goal of continuous relaxation problems is to find a set of **preferred** and **minimal** continuous relaxations that can resolve the conflicts in an inconsistent OCSTN. However, continuous relaxation problems are different from discrete relaxation problems in that their solution spaces are infinite. The discrete relaxation to a temporal constraint only has two states: preserved and suspended. Therefore, the number of possible discrete relaxations to a OCSTN is countable, though exponential in the size of constraints.

On the other hand, the continuous relaxation to a temporal constraint has an infinite number of states: its temporal bounds can be relaxed to any consistent pair of numbers in $\mathcal{R}$. For example, (Figure 4-1) shows three sample continuous relaxations to a temporal constraint. As a result, the number of continuous relaxations to an inconsistent OCSTN is infinite, making it difficult to apply the concept of minimality and preference models we defined in Chapter 3.



(a) The original constraint



(b) Sample relaxation 1 (c) Sample relaxation 2 (d) Sample relaxation 3

Figure 4-1: Examples of continuous relaxation to a temporal constraint

In this section, we address these problems by proposing a new preference model and definition of continuous temporal relaxations, such that:

- The preference models can be used to compare two or more continuous relaxations in terms of their modified constraints and degrees of relaxation.

- A minimal continuous relaxation can be used to represent a set of continuous relaxations. Then, a finite set of minimal continuous relaxations can represent the infinite number of continuous relaxations to an inconsistent OCSTN.

The continuous relaxation problem is formally defined as:

Definition 38. *(Continuous Relaxation Problem) Given an inconsistent OCSTN $\mathcal{P}$ and a user preference model UPM, a* **Continuous Relaxation Problem** *is a problem of finding a continuous relaxation, CR, such that:*

- *CR makes $\mathcal{P}$ consistent.*
- *CR is the most preferred continuous relaxation according to UPM.*
- *CR is a minimal continuous relaxation.*

We will define the term *preferred* and *minimal* precisely in the following subsections, building upon the definitions of minimal and preferred discrete relaxations given in Chapter 3.

4.1.1 Continuous Preference Models Over Temporal Constraints

This subsection defines the preference models that can be used to enumerate preferred continuous relaxations. In Chapter 3, we defined the preference model for discrete relations, which specifies the user's intent towards the suspension of temporal constraints. It is a preference model over a finite set of discrete states. For continuous relaxation, the preference model has to cover an infinite set of states: it must specify the user's willings towards different degree of relaxation to a simple temporal constraint. The domain of a simple temporal constraint can be relaxed to any valid pair of real numbers, and the preference model for continuous relaxations must cover a continuous domain.

Definition 39. *(Preference Models over Continuous Relaxations) A preference model, UPM, for an inconsistent OCSTN, $\mathcal{P}$, must satisfy:*

- *Domain: UPM is defined over the complete scope of $\mathcal{P}$. That is, UPM can be used to evaluate any relaxation to any temporal constraints in $\mathcal{P}$, including the combinations of relaxations and choices made to the decision events in $\mathcal{P}$.*

- *Comparison: Given any two continuous relaxations, CR and CR', to $\mathcal{P}$, UPM can be used to choose the one that is more preferred by the user, if not equally preferred.*

- *For any temporal constraints in $\mathcal{P}$, the users would always prefer no relaxation than any temporal relaxations.*

In [32], several examples of preference models over temporal constraints are provided, including linear, step and quadratic functions. Its model of semi-convex preference functions over temporal constraints is adopted by Uhura to represent the cost of continuous relaxations over temporal constraints. The semi-convex functions involve linear, convex and step functions: the outputs monotonically increase on the upper bound of the simple temporal constraints, and monotonically decrease on the lower bound. Therefore, the original constraints without any relaxation will always cost zero.

The semi-convex preference functions satisfy all three requirements for the preference model of continuous relaxations. First, the range of such a function is defined over $\mathcal{R}$, hence covers all continuous relaxations to temporal constraints. Second, these functions return a metric value for each evaluation of temporal constraints so that two continuous relaxations can be compared. Finally, given that the semi-convex functions are monotonically decreasing on the lower bound and increasing on the upper bound of any temporal constraints, any continuous relaxations will incur a cost higher than no relaxation, whose is always zero. In this thesis, we define a continuous metric cost function over continuous relaxations to simple temporal constraints. It is a semi-convex function constructed using linear functions.

Definition 40. *(Continuous metric cost function of continuous relaxations) Let CR_k be a continuous temporal relaxation that relaxes a simple temporal constraint, stc_k,*

to stc'_k. The cost of CR_k is defined through a function $f(stc_k, stc'_k) \rightarrow \mathbb{R}$ that maps CR_k to a cost where:

- $f(stc_k, stc'_k) = C_{LB}+C_{UB} = f_{LB}(LB(stc_k), LB(stc'_k))+f_{UB}(UB(stc_k), UB(stc'_k))$. *$f_{LB}$ and f_{UB} are* **linear** *functions that map the difference between the lower bounds and upper bounds of stc_k and stc'_k to two real values, C_{LB} and C_{UB}, representing the costs of relaxing the lower bound and upper bound.*
- $f_{LB}(lb, lb') = a_{LB}(lb' - lb) + b_{LB}$ *where* a_{LB} *and* $b_{LB} \in \mathbb{R}^- \cup 0$.
- $f_{UB}(lb, lb') = a_{UB}(ub' - ub) + b_{UB}$ *where* a_{LB} *and* $b_{LB} \in \mathbb{R}^+ \cup 0$.
- $BC = b_{LB} + b_{UB}$ *is called the basic cost of $\mathcal{CR}_k$, which is the lower bound of the relaxation cost.*

In (Figure 4-2), several examples of preference functions are presented. The bold parts in the graphs of the functions represent the original span of the temporal constraints. Figure 4-2 (a), (b) and (c) are considered as semi-convex while (d) and (e) are not. Among these three, only (b) fits our definition of preference model: the cost must be linear with regards to the degree of relaxation. Neither (a) nor (c) has a linear relationship between cost and the relaxation to lower/upper bound. The metric cost function we used is a simple type of semi-convex preference functions, in which the functions on the lower bound and upper bound sides are both linear. We assume that a_{LB} is always negative, and a_{UB} is always positive. Therefore, the metric cost function we defined is semi-convex, which implies that the user prefers strictly smaller relaxations to the upper and lower bounds of the temporal constraints.

Similar to the metric cost functions for discrete relaxations, the cost of a continuous relaxation to an inconsistent OCSTN is the sum of the costs of all relaxed temporal constraints, plus the cost of decisions. It represents the user's intent towards the relaxation of multiple constraints as well as the decisions made by the algorithm.

Definition 41. *(Cost of continuous relaxations) Let CR be a continuous relaxation to an inconsistent OCSTN, $\mathcal{P}$. Its cost is defined as* $\textsc{Cost}(CR) = C_{rlx} + C_{dec}$ *where:*

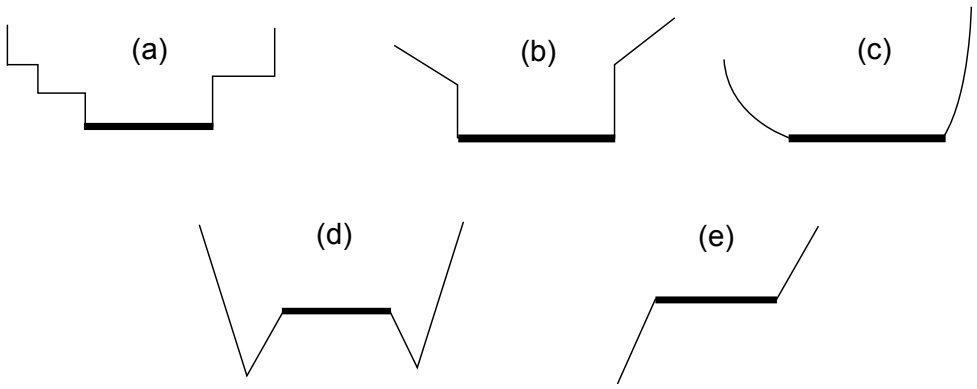


Figure 4-2: Examples of semi-convex preference functions (a)-(c) and non-semi-convex functions (d)-(e)

- $C_{rlx} = c_{rlx1} + c_{rlx2} + ... + c_{rlxm}$ *is the sum of the costs of all continuously relaxed temporal constraints.*

- $C_{dec} = c_{dec1} + c_{dec2} + ... + c_{decn}$ *is the sum of the costs of all decisions in* CR.

The discrete metric cost function presented in Chapter 3 can be viewed as a special case of the continuous metric cost function. The cost of the discrete relaxation to a temporal constraint is a fixed value, since there is one and only one possible discrete relaxation. By setting the a_{LB} and a_{UB} to zero, the value of a continuous metric cost function is fixed to $b_{LB} + b_{UB}$, which is its basic cost.

Example: Continuous Relaxation of John's Trip

We demonstrate the difference between discrete and continuous relaxations in this subsection. In Chapter 3, we presented an over-constrained scenario of John's trip from office to home: he would like to stop by a sandwich restaurant for dinner and then arrive home in 60 minutes. The planner generated a TPN that encodes six different sequences of activities that can satisfy his requirements about food and destinations. However, none of the plans are temporally consistent, since they all require more than 60 minutes. The corresponding OCSTN is shown in (Figure 4-3).

In the previous chapter, we described two relaxations to the inconsistent OCSTN: suspending the *TimeConstraint [0min,60min]* or *(dine-in Cosi) [15min,15min]* (Figure 4-5(a) and 4-5(b)). The latter one is preferred since the cost of suspending the

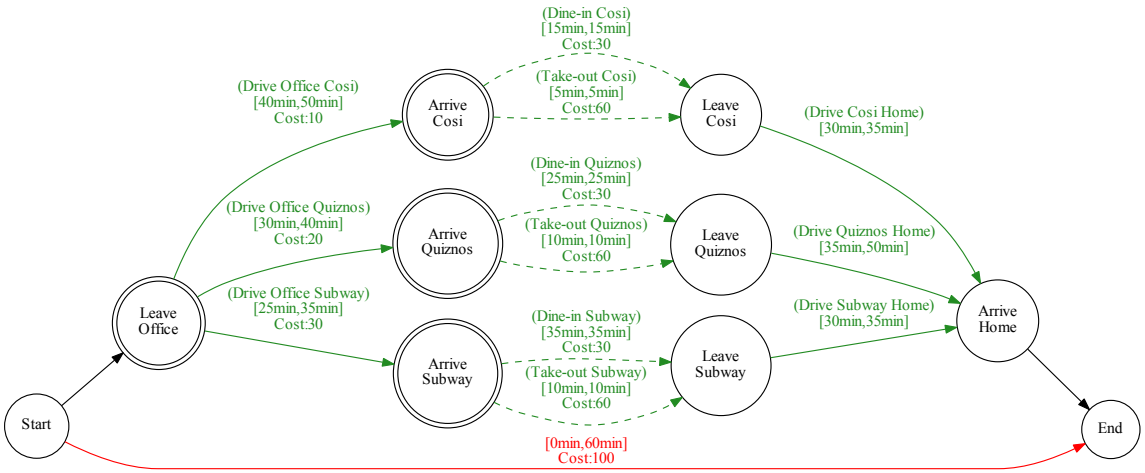


Figure 4-3: An inconsistent OCSTN

temporal constraint (dine-in Cosi) is much less than the former one. Neither option sounds ideal to John due to the nature of discrete relaxations: the temporal constraints, which represent users' goals, are removed completely. In such a situation, continuous relaxations provide a better resolution that minimizes the perturbation to John's plans.

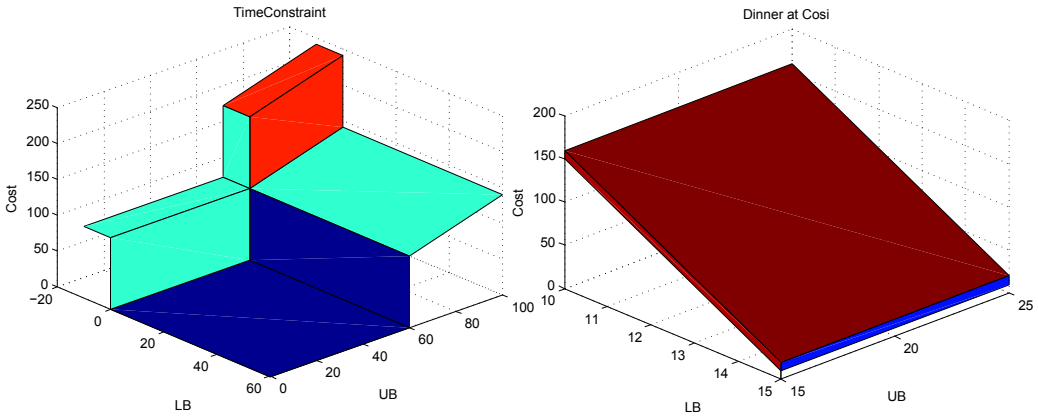
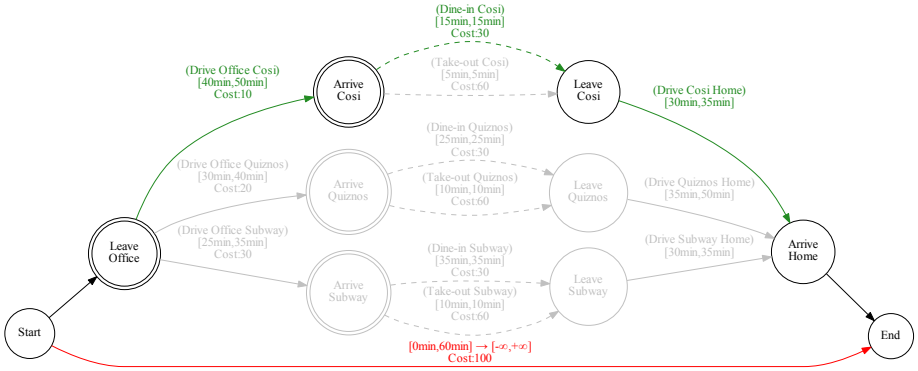


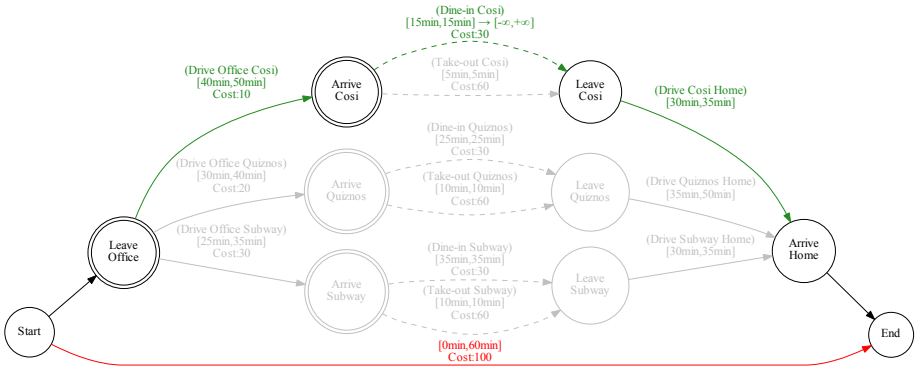
Figure 4-4: Cost functions over constraints **TimeConstraint** and **Dinner at Cosi**

Assume that the preference functions over the relaxations of both constraints, *TimeConstraint* and *(dine-in Cosi)*, are (Figure 4-2). It can be seen from the graph that John is not willing to increase the *TimeConstraint*, since the basic cost of changing the temporal bounds is already 100, and keeps increasing with the increase of the upper bound. On the other hand, he is ok if the dinner time is extended, since the

cost of extending the dinner time is only 10 and does not increase regardless of the increase of the upper bound. However, John will be very unhappy if the dinner time gets reduced, since the cost of lowering the lower bound of his dinner time increases rapidly.



(a) A discrete relaxation that suspends *TimeConstraint*

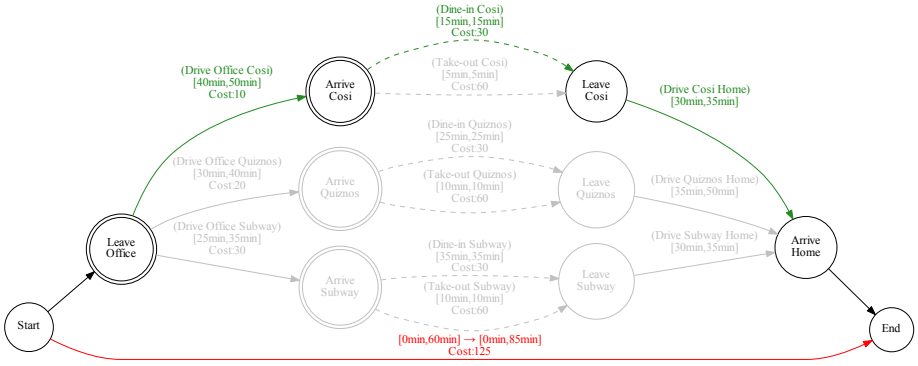


(b) A discrete relaxation that suspends *(Dine-in Cosi)*

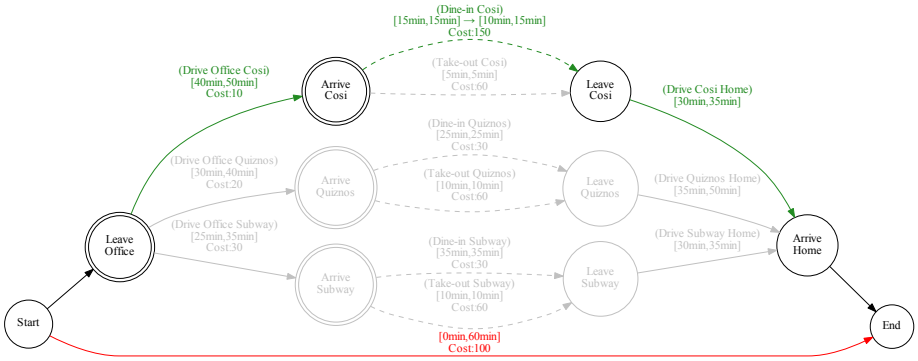
Figure 4-5: The discrete temporal relaxations to John's trip

Two continuous relaxations to John's problem are shown in (Figure 4-6(a) and 4-6(b)): the *TimeConstraint* constraint has to be relaxed to 85 minutes, while the *(dine-in Cosi)* constraint is shorten to 10 minutes.

According to the preference function, we can evaluate the costs of both continuous relaxations. Relaxing the overall temporal constraint costs less than shorten the dinner time: the cost of relaxing the upper bound of *TimeConstraint* to 85 is 125, while the cost of relaxing the lower bound of *(dine-in Cosi)* is 150 (Figure 4-7). Therefore, the continuous relaxation to *TimeConstraint* is a better resolution for John.



(a) A continuous relaxation of *TimeConstraint*



(b) A continuous relaxation of *(dine-in Cosi)*

Figure 4-6: The continuous temporal relaxations to John's trip

4.1.2 Minimal Continuous Temporal Relaxations to OCSTNs

This subsection defines *minimal* continuous relaxations to an inconsistent OCSTN. We present this concept as a generalization to the minimal discrete relaxations: in addition to the minimality in terms of the relaxed temporal constraints, a minimal continuous relaxation also makes minimal modifications to each temporal constraints. As stated before, the goal is to use a finite set of minimal continuous relaxations to represent the infinite number of continuous relaxations to an inconsistent OCSTN.

Recall from chapter 3 that minimal discrete relaxations are computed as the minimal covering sets of all minimal conflicts in an inconsistent OCSTN. A discrete relaxation is minimal in that if any suspension of temporal constraints is removed from it, the relaxation is no longer consistent. Therefore, minimal discrete relaxations is a compact representation of *all* valid discrete relaxations to an inconsistent OCSTN.

For example, (Figure 4-8(a)) shows an inconsistent STN with two temporal con-

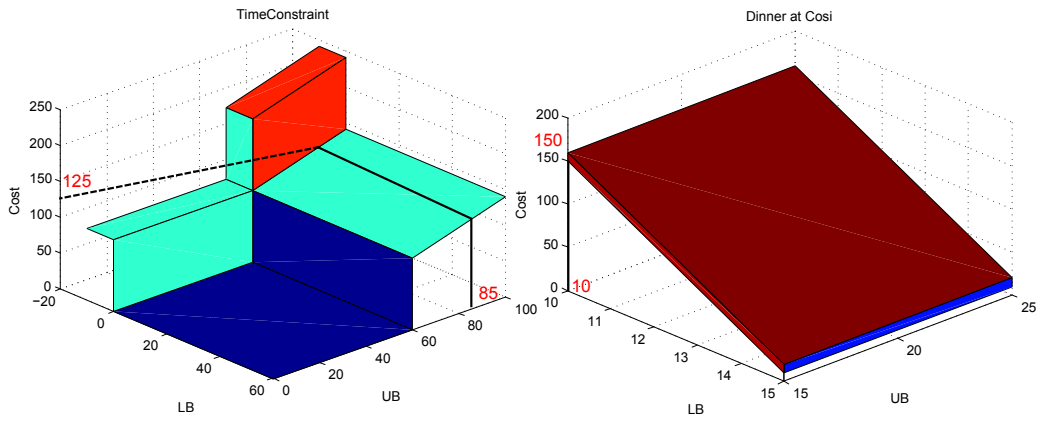
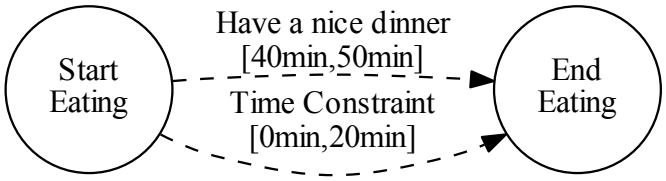
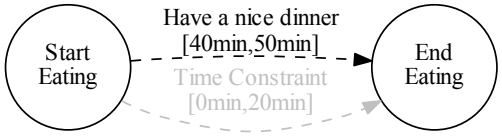


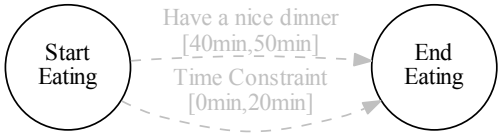
Figure 4-7: Cost after continuously relaxing *TimeConstraint* and *(dine-in Cosi)*



(a) An over-constrained temporal problem



(b) A minimal discrete relaxation



(c) A non-minimal discrete relaxation

Figure 4-8: Examples of discrete relaxations

straints and two events. (Figure 4-8(b)) is one of its minimal discrete relaxations that suspends the *Time constraint [0min,20min]*. This relaxation is minimal in that if *Time constraint [0min,20min]* is not suspended, the STN will be inconsistent. On the other hand, (Figure 4-8(c)) shows a non-minimal discrete relaxation. It suspends both temporal constraints in the problem, which is unnecessary, since suspending one of them is enough for making the STN consistent.

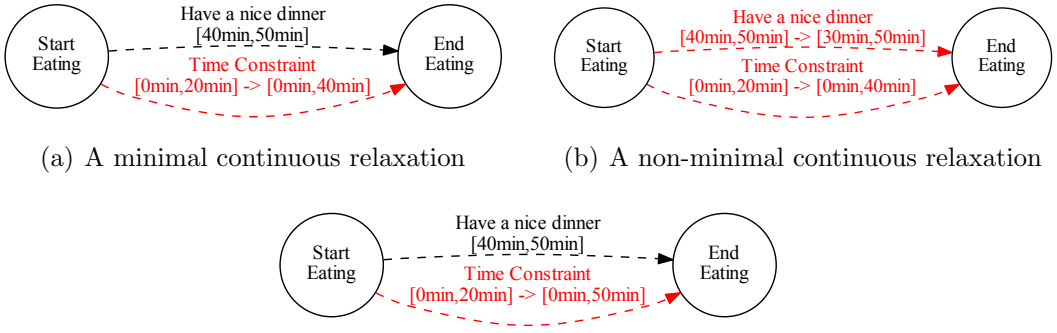
We define **minimal continuous relaxations** to an inconsistent OCSTN based on minimal discrete relaxations.

Definition 42. *(Minimal continuous relaxation) A minimal continuous relaxation, MCR, is a continuous relaxation to an inconsistent OCSTN, $\mathcal{P}$, where:*

- *If MCR relaxes a set of temporal constraints, TCs, then there does NOT exist a continuous relaxation, MCR', that only relaxes a proper subset of TCs but makes $\mathcal{P}$ consistent. MCR satisfies the requirements of a minimal discrete relaxation.*

- *If MCR relaxes the temporal bounds of a set of temporal constraints, TCs, to $TBs = [lb_1, ub_1], [lb_2, ub_2], ..., [lb_k, ub_k]$, then there does NOT exist a continuous relaxation, MCR'', that relaxes the temporal bounds of TCs to a set of narrower bounds, $TBs' = [lb_1 + \delta_l b1, ub_1 - \delta_u b1], [lb_2 + \delta_l b2, ub_2 - \delta_u b2], ..., [lb_k + \delta_l bk, ub_k - \delta_u bk]$, but makes $\mathcal{P}$ consistent. $\delta_l bi \geq 0, \delta_u bi \geq 0$ and $\sum_{1...k} \delta_l bi + \delta_u bi > 0$.*

The first criteria is similar to that of discrete relaxations: none of the subsets of a minimal continuous relaxation resolves all conflicts in the inconsistent problems. For example, in (Figure 4-9), the continuous relaxation that relaxes both *Have a nice dinner* and *Time constraint* is the superset of the other continuous relaxation, which only relaxes *Time Constraint*. Therefore, (Figure 4-9(b)) is not a **minimal** continuous relaxation.



(a) A minimal continuous relaxation

(b) A non-minimal continuous relaxation

(c) Another non-minimal continuous relaxation

Figure 4-9: Examples of continuous relaxations

The second criteria requires reasoning on temporal bounds: a minimal continuous relaxation must minimally relaxes the temporal constraints in an inconsistent temporal problem. In other words, there is no unnecessary relaxations of temporal bounds: if a conflict can be resolved by relaxing $[a, b]$ to $[a, b + 10]$, then any continuous re-

laxations that relaxes the constraint to $[a, c]$ where $c > b + 10$ are not considered minimal.

For example, (Figure 4-9(c)) is not considered as a minimal continuous relaxation, even though it only relaxes one temporal constraint: there is another relaxation set (Figure 4-9(a)) whose continuous relaxation provides a tighter bound ([0min,40min] vs [0min,50min]).

Therefore, we can use a finite set of minimal continuous relaxations to represent the infinite number of continuous relaxations to an inconsistent OCSTN. In fact, given a minimal discrete relaxation, we can find a minimal continuous relaxation by minimally relaxing the suspended constraints. Hence the number of minimal continuous relaxations to an inconsistent OCSTN is equal to that of minimal discrete relaxations. By using this property in the enumeration of minimal continuous relaxations, we can continue to use the methods we developed for discrete BCDR, including conflict extraction and updating candidates.

Lemma 1. *[Number of Minimal Continuous Relaxations] Given an inconsistent OCSTN, $\mathcal{P}$, the number of minimal continuous relaxations to $\mathcal{P}$ is equal to that of minimal discrete relaxations. For any minimal discrete relaxation MDR to $\mathcal{P}$, we can find a minimal continuous relaxation MCR by minimally relaxing the suspended constraints in MDR.*

4.2 Conflict-directed Enumeration of Continuous Relaxations

In this section, we present the continuous relaxation algorithm, CONTINUOUS BCDR, that has been integrated with Uhura to support the enumeration of continuous relaxations. We develop CONTINUOUS BCDR based on the discrete version of Best-first Conflict-directed Relaxation, with new continuous preference models and conflict resolution techniques. We will continue to use the conflict extraction method introduced in Chapter 3. We first give an overview of the algorithm in this section. The details of the conflict resolution and candidate generation functions are presented in Section 4.3 and 4.4.

4.2.1 An Overview of Continuous BCDR

The Continuous BCDR algorithm is shown in (Algorithm 7). It takes in an inconsistent OCSTN, detects the conflicts and generates preferred continuous relaxations. Similar to the enumeration of discrete relaxations, there are four major steps in CONTINUOUS BCDR.

- Generate candidate: select the most preferred candidate continuous relaxation from all available ones.
- Check consistency: given a candidate relaxation, check if it resolves all conflicts in the inconsistent temporal problem.
- Extract conflicts: if a candidate fails the consistency check, extract minimal conflicts from the candidate.
- Extend candidate: generate candidate continuous relaxations from the known conflicts and update all existing candidates with newly discovered minimal conflicts.

The first two steps are identical to those in the generation of discrete relaxations. Function CHECKCONSISTENCY checks the consistency of candidate continuous relax-

input : *OCSTN*, an inconsistent OCSTN
input : *UPM*, the user preference model associated with the temporal constraints and decision events in *OCSTN*
input : *K*, number of minimal relaxation sets needed
output: *MCRs*, *K* continuous minimal relaxation sets to *OCSTN* or all available discrete minimal relaxation sets, whichever is larger.

```
// Initialize.
```

1 $CANDs \leftarrow \{$BestContinuousCandidate($UPM, OCSTN$)$\}$: Generate the best candidate.;
2 $MCRs \leftarrow \{\}$: No results generated yet;
3 $CFLTs \leftarrow \{\}$: No conflicts found yest;
4 $i = 0$: Reset result counter;

```
// Generate new candidate and test until the maximum number is
   reached, or run out of candidates.
```

5 **while** $i < K$ **do**
6 $currCand \leftarrow$ DequeueBestContinuousCandidate($CANDs, OCSTN, UPM$);

```
// If consistent, record the current candidate; Otherwise
   extract new conflict and update existing candidates
```

7 **if** ConsistencyCheck*(currCand)* **then**
8 $MCRs \leftarrow MCRs \cup currCand$;
9 $i++$;
10 **else**
11 $CFLTs \leftarrow CFLTs \cup$ ExtractMinConflict($currCand$);
12 $CANDs \leftarrow$ UpdateContinuousCandidates($CFLTs, CANDs$);
13 **end**

```
// If all candidates have been checked and are consistent,
   no more relaxation sets can be generated.  Return the
   MCRs.
```

14 **if** $CANDs = MCRs$ **then**
15 **return** *MCRs*;
16 **end**
17 **end**
18 **return** *MCRs*;

Algorithm 7: Continuous BCDR

ations using negative loop detection algorithms. If inconsistent, function EXTRACT-MINCONFLICT will extract minimal conflicts from the candidate by splitting the negative loops. These methods have been presented in Chapter 3.

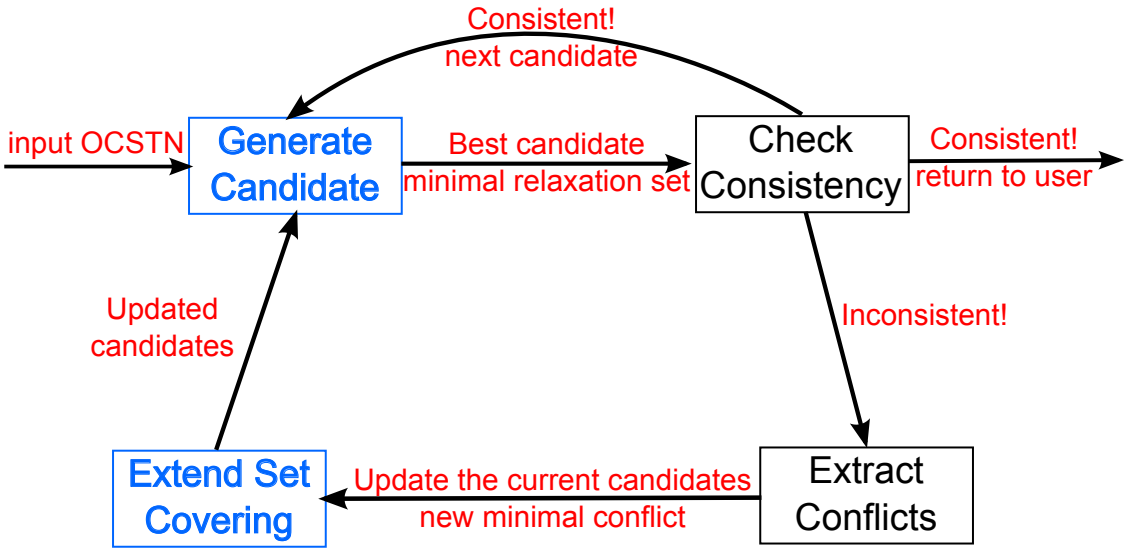


Figure 4-10: The program flow of CONTINUOUS BCDR

The third and the fourth steps are different between the enumerations of discrete and continuous relaxations. They are highlighted in (Figure 4-10). For the third step, we use a different strategy for conflict resolution. To resolve a minimal conflict using discrete relaxation, we may simply suspend one temporal constraint in the conflict. On the other hand, for continuous relaxation, we have to minimally relax the temporal bounds of one constraint until the conflict is resolved.

For the fourth step, the process of evaluating preference between candidate relaxations is different. For discrete relaxation, each temporal constraint has only two states: preserved and suspended. The user preferences over each state are encoded explicitly in the preference models using real numbers (metric cost functions). For continuous relaxations, each simple temporal constraint may be relaxed to any pair of real numbers. We need to compute the cost using the simple semi-convex functions defined through the preference model. Finding the best candidate continuous relaxation is more like optimizing a *Linear Programming* problem, in which both

constraints and utilities are linear functions over variables.

4.2.2 Proving the Correctness of Continuous BCDR

The *Continuous BCDR* algorithm is guaranteed to find all minimal continuous relaxations in best-first order, given a user preference model. We state two lemmas here in order to prove the completeness and soundness of *Continuous BCDR*. We use these two lemmas in the proof of the CONTINUOUS BCDR algorithms. Both lemmas will be proved later in Section 4.3.

Lemma 2. *[Completeness of* UPDATECONTINUOUSCANDIDATES*] (Algorithm 7, 9) Given a set of minimal conflicts, $MinCFLTs$, function* UPDATECONTINUOUSCANDIDATES *generates* **all** *candidate continuous relaxations to $MinCFLTs$.*

Lemma 3. *[Soundness of* UPDATECONTINUOUSCANDIDATES*] (Algorithm 7, 9) Given a set of minimal conflicts, $MinCFLTs$, the candidate continuous relaxations generated by function* UPDATECONTINUOUSCANDIDATES *are all valid candidate continuous relaxations. That is, all the candidates can resolve all known conflicts in $MinCFLTs$.*

First, we show that CONTINUOUS BCDR is complete in that it generates all minimal continuous relaxations given an inconsistent OCSTN.

Theorem 7. *[Completeness of* CONTINUOUS BCDR*] Given an inconsistent OCSTN, $\mathcal{P}$,* CONTINUOUS BCDR *can find all minimal continuous relaxations, $MCRs$, to $\mathcal{P}$.*

Proof. [Proof by contradiction] Assume that CONTINUOUS BCDR generate a set of minimal continuous relaxations, $MCRs$, that missed one minimal continuous relaxation, $MCR*$, to $\mathcal{P}$. By Lemma 2, if one minimal relaxation is not generated by UPDATECONTINUOUSCANDIDATE, then there must be a minimal conflict, $MinCFLT$, that has not been detected by CONTINUOUS BCDR yet.

Therefore, there must be at least one candidate continuous relaxation generated by UpdateContinuousCandidate that cannot resolve $MinCFLT$. However, it contradicts our assumption: all the minimal continuous relaxations in $MCRs$ can resolve all conflicts in $\mathcal{P}$. Hence the assumption does not hold: Continuous BCDR generates all minimal continuous relaxations given an inconsistent OCSTN. □

Next, we demonstrate that Continuous BCDR is sound in that all the minimal continuous relaxations generated are valid, that is, are *minimal* and can resolve all the conflicts given an inconsistent OCSTN.

Theorem 8. *[Soundness of* Continuous BCDR*] Given an inconsistent OCSTN, $\mathcal{P}$, all the minimal continuous relaxations, $MCRs$, generated by* Continuous BCDR *resolve $\mathcal{P}$.*

Proof. A minimal continuous relaxation generated by Continuous BCDR, MCR, is valid in two aspects:

- MCR must resolve all conflicts in $\mathcal{P}$. Since MCR passes the consistency check (Function ConsistencyCheck), it must resolve all the conflicts in the inconsistent OCSTN.
- MCR must be minimal. By Lemma 3, all candidate continuous relaxations generated by UpdateContinuousCandidates are minimal. Further, all the continuous relaxations are generated by UpdateContinuousCandidates. Therefore, MCR is a minimal continuous relaxation.

Therefore all the minimal continuous relaxations, $MCRs$, generated by Continuous BCDR resolve $\mathcal{P}$. □

In summary, the Continuous BCDR algorithm can be viewed as the discrete BCDR algorithm with new conflict resolution and candidate selection techniques. It resolves continuous relaxation problems by enumerating minimal continuous relaxations in best-first order. Like the discrete version of BCDR, Continuous BCDR

is also complete and sound in that it can find all minimal continuous relaxations to an inconsistent OCSTN, and guarantees the correctness of the results. This algorithm is made incremental so that the relaxations can be returned prior to the detection of all conflicts.

4.3 Generating Candidates from Conflicts

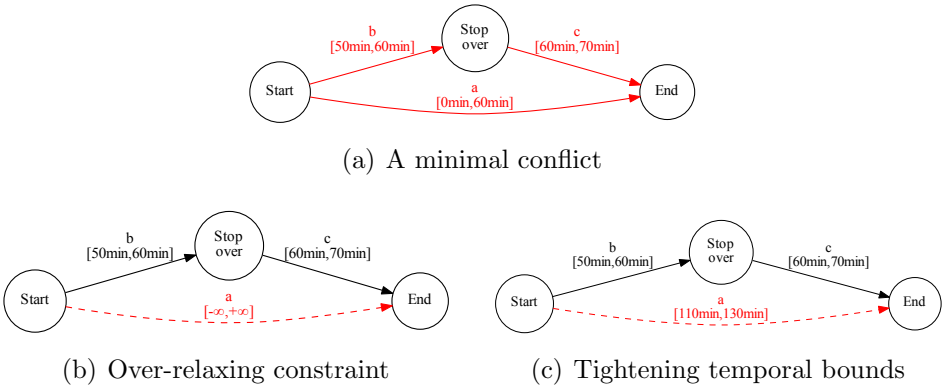
This section presents the third step in CONTINUOUS BCDR: given a set of minimal conflicts, $MinCFLTs$, generate candidate minimal continuous relaxations that can resolve $MinCFLTs$. This is achieved through two functions. First, function CONTINUOUSLYRESOLVECONFLICT generates constituent continuous relaxations that resolve each minimal conflict individually. Then we generate minimal relaxations that resolve all conflicts by combining the constituent relaxations. This is similar to the DISCRETE BCDR and CD-A* algorithm. The process is made incremental by function UPDATECONTINUOUSCANDIDATES so that we do not have to recompute all constituent relaxations when a new conflict is detected, which is the same as DISCRETE BCDR.

4.3.1 Resolving Conflicts Using Constituent Relaxations

To resolve a minimal conflict, it is necessary and sufficient to fully relax one constraint in order to resolve that conflict. The individual constraint that resolves the conflict are called constituent relaxations of that conflict. As we stated in Section 4.1, a discrete relaxation can be viewed as a special case of a continuous relaxation, and suspending one constraint is equivalent to relaxing its temporal bound to $[-\infty,+\infty]$. Because continuous relaxation problem requires the minimal amount of modification made to the temporal constraints, we have to compute the tightest temporal bounds for the temporal constraints that can resolve the conflicts.

We address this challenge using a 2-step approach: first, we *over-relax* the conflict by generating discrete constituent relaxations to it. Then we check each constituent relaxation and compute the tightest bound for it. For example, (Figure 4-11) shows an example of such a relaxation process. The temporal bound of constraint **a** is first over-relaxed (Figure 4-11(b)). Then the continuous constituent relaxation of **a** is computed based on constraints **b** and **c** (Figure 4-11(c)).

This procedure is implemented in (Function CONTINUOUSLYRESOLVECONFLICT Algorithm 8). We first change the temporal bounds of one temporal constraint in



(a) A minimal conflict

(b) Over-relaxing constraint

(c) Tightening temporal bounds

Figure 4-11: Two steps in the generation of constituent relaxations

the minimal conflict to $[-\infty, +\infty]$ (Line 2). By now, the minimal conflict has been resolved due to the property of minimal conflicts: suspending any temporal constraint in a minimal conflict will resolve it.

Next, we tighten the over-relaxed constraint, since a feasible temporal bound smaller than $[-\infty, +\infty]$ always exists. ContinuouslyResolveConflict uses the Floyd-Warshall algorithm (All-Pair-Shortest-Path) to tighten the temporal bound of the relaxed temporal constraint (Line 3-10). Floyd-Warshall checks all temporal constraints in the minimal conflict and computes the tightest temporal bounds of the over-relaxed constraint.

Note that we are relaxing only one temporal constraint in each constituent relaxation to a minimal conflict. This is because of the requirements on the minimality of relaxations: Continuous BCDR only generates minimal continuous relaxations. If a conflict can be resolved by relaxing one constraint, we will **not** consider any relaxations with two or more relaxed constraints. This may bring a problem when the user is looking for an optimal resolution, since slightly relaxing two constraints may be more preferred compared to relaxing one constraint by a lot. We address this problem in Section 4.4.

4.3.2 Incrementally Updating Candidate Relaxations

Next, we combine the constituent relaxations generated in the previous step into candidate minimal relaxations. The problem can be defined as: Given a set of con-

input : *NewMinCFLT*, newly discovered minimal conflict
output: *ConstituentRelaxations*, a set of constituent relaxations that can resolve the minimal conflict

```
// Generate continuous constituent relaxations of the minimal conflict
for Constraint in RelaxableConstraint(NewMinCFLT) do
    // Reset the temporal bounds of the relaxable constraint to [-infinity,+infinity]
    ResetTemporalBounds(Constraint);
    // Compute the tightest feasible bound of Constraint using Floyd-Warshall
    for M in Events(NewMinCFLT) do
        for S in Events(NewMinCFLT) do
            for E in Events(NewMinCFLT) do
                UB(S,E) = Min(UB(S,E), UB(S,M) + UB(M,E));
                LB(S,E) = Max(LB(S,E), LB(S,M) + LB(M,E));
            end
        end
    end
    ConstituentRelaxations ← ConstituentRelaxations ∪ Constraint
end
return ConstituentRelaxations
```

Algorithm 8: ContinuouslyResolveConflict

stituent relaxations, $CR_1, CR_2, ..., CR_n$, to a set of minimal conflicts, $MinCFLTs = conflict_1, conflict_2, ..., conflict_n$, generate a set of candidate minimal continuous relaxations, $CandMCRs$, that resolves all conflicts in $MinCFLTs$.

This can be viewed as a minimal set covering process. Each candidate relaxation $CandMCR$ must resolve **all** conflicts in $MinCFLTs$. The candidate relaxation contains at least one constituent relaxation from each set in $CR_1, CR_2, ..., CR_n$. Moreover, we would like $CandMCR$ to be minimal so that none of the constituent relaxations can be removed from $CandMCR$ without making it inconsistent.

Theorem 9. *The minimal covering sets (hitting sets) of the constituent relaxations, $CR_1, CR_2, ..., CR_n$, to a set of minimal conflicts, $MinCFLTs = conflict_1, conflict_2, ..., conflict_n$, are the minimal continuous relaxations to $MinCFLTs$.*

This property is presented as the duality between minimal conflicts and relaxations in [10, 4, 34]. We use a similar approach in Chapter 3 to generate candidate discrete relaxations from known conflicts. However, there is a difference between discrete and continuous relaxations in the process of combining constituent relaxations. For discrete relaxation, if two constituent relaxations suspends the same temporal constraint, tc_k, then only one constituent relaxation needs to be combined into the relaxations.

For continuous relaxations, even though two constituent relaxations, cr_a and cr_b, relax the same temporal constraint, tc_k, they may assign different relaxed bounds to tc_k. Therefore, when combining two constituent relaxations like cr_a and cr_b, we have to merge their temporal bounds from $[lb_a, ub_a]$ and $[lb_b, ub_b]$ to [MIN(lb_a, lb_b),MAX(ub_a, ub_b)], such that the resulting continuous relaxation is guaranteed to resolve all conflicts.

In the Chapter 3, we introduced an incremental minimal set covering method used by BCDR: the candidate relaxations can be updated using newly detected conflicts without re-computing the constituent relaxations to all known conflicts. We use the same approach to compute continuous candidates in CONTINUOUS BCDR (Function UPDATECONTINUOUSCANDIDATES, Algorithm 9). It generates all combinations of constituent relaxations that can resolve all known conflicts (Line 2), then minimize

the candidates by removing all redundant continuous relaxations (Line 3).

input : *NewMinCFLT*, newly discovered minimal conflict
input : *PrevContCandidates*, previous candidate minimal relaxation sets
output: *UpdatedContCandidates*, updated candidate minimal relaxation sets

```
// Generate constituent relaxations of the minimal conflict
```
1 *ConstituentRelaxations* ← ContinuouslyResolveConflict(*NewMinCFLT*);
```
// Start with the cross product of previous candidates and the
   new conflict.
```
2 *UpdatedContCandidates* ← *PrevContCandidates* ⊗ *ConstituentRelaxations*;
```
// Remove redundant (non-minimal) candidates.
```
3 *UpdatedContCandidates* ← RemoveRedundantCandidates(*UpdatedContCandidates*);
4 **return** *UpdatedContCandidates*

Algorithm 9: UpdateContinuousCandidates

In Section 4.2 we present two Lemmas of the completeness and soundness of Function UpdateContinuousCandidates to prove the completeness of Continuous BCDR. Here we present the proof of these Lemmas. First, we show that UpdateContinuousCandidates is complete in that it generates all candidate minimal relaxations given a set of conflicts.

Proof. [Completeness of UpdateContinuousCandidates. Proof by contradiction]

Assume that given a set of minimal conflicts, $MinCFLTs$, UpdateContinuousCandidates generates all candidate minimal continuous relaxations except for one, $CandMCR*$.

Since the minimal set covering procedure is complete and sound [4], missing $CandMCR*$ is the result of the conflict resolution procedure. In other words, at least one constituent relaxation, $cr*$ to the conflict $conflict_k$, is not generated by Function ContinuouslyResolveConflict.

However, ContinuouslyResolveConflict loops through every temporal constraint in $conflict_k$ to generate constituent relaxations. There is no possibility that one temporal constraint is skipped that leads to the missing of $cr*$.

Therefore the assumption does not hold. UpdateContinuousCandidates is complete.

□

Second, we show that all candidates generated by UpdateContinuousCandidates are valid *minimal* continuous relaxations that can resolve *all* known conflicts.

Proof. [Soundness of UpdateContinuousCandidates]

Given a set of minimal conflicts, $MinCFLTs$, the candidate continuous relaxation generated by UpdateContinuousCandidates, $CandMCR*$, is valid in two aspects:

- $CandMCR*$ is minimal. This is guaranteed by the minimal set covering procedure [4]: all candidates are minimal covering set of the constituent relaxations.
- $CandMCR*$ resolves all conflicts in $MinCFLTs$. According to the property of covering sets, $CandMCR*$ contains at least one constituent relaxation to each minimal conflict in $MinCFLTs$. Hence, all conflicts in $MinCFLTs$ can be resolved by $CandMCR*$.

Therefore UpdateContinuousCandidates is sound.

□

In summary, this section presented an innovative method that can generate minimal continuous relaxations from the conflicts in an inconsistent OCSTN. Given a minimal conflict, it can be resolved by continuously relaxing one of its constraints. We use the Floyd-Warshall algorithm to compute the tightest bound of each relaxed constraint. Given a set of minimal conflicts in an inconsistent temporal problem, we generate all minimal continuous relaxations to it by computing the minimal covering sets of the constituent relaxations to each conflict. By using an incremental set covering algorithm, we can update current candidates with newly detected conflicts without recomputing all constituent relaxations.

4.4 Generating Preferred Continuous Relaxation Candidates

In this section, we present the method that finds the best candidate from all candidate continuous relaxations generated in the previous step: Given a list of candidate continuous relaxations to an inconsistent OCSTN, $MCRs$, and a preference model, UPM, select the most preferred candidate $MCR* \in MCRs$. Further, we will prove that the $MCR*$ generated using this 2-step approach, first generating all candidate relaxations then selecting the most preferred candidate from the collection, is also the best candidate continuous relaxation that can resolve all known conflicts.

4.4.1 Selecting the Most Preferred Candidate

To select the most preferred candidate continuous relaxation from a list of candidates, we use a similar approach to the one that is used in discrete relaxations: iterating through all candidates in the list, evaluating the cost of each candidate using the continuous preference models, and selecting the one with the lowest cost through binary comparisons. The procedure is implemented in Function DequeueBestContinuousCandidate (Algorithm 10).

DequeueBestContinuousCandidate takes in a set of candidate continuous relaxations, $CANDs$, and returns the most preferred one, *BestContinuousCandidate*, according to a continuous preference model *cUPM*. It starts by randomly selecting a candidate as the *Currently Best* one (Line 1) and recording its utility as the currently lowest cost (*LeastCost*, Line 2). Next, DequeueBestContinuousCandidate iterates through all candidates in $CANDs$ (Line 3): if a candidate *currCandidate* costs less than *LeastCost* (Line 5), *currCandidate* will be recorded as the 'Currently Best' candidate (Line 6) and its cost is recorded as *LeastCost* (Line 7).

The Function GetCost evaluates each continuous relaxation using the utility functions defined in the preference model (Algorithm 11). It computes the cost of each temporal constraint relaxed by CR using the linear preference functions over

```
input  : CANDs, a list of candidate continuous minimal relaxation sets
input  : cUPM, the continuous user preference model
output: BestContinuousCandidate, the best candidate in CANDs

// Initialize BestCandidate with the first candidate in the
   list and the incumbent value.
1  BestContinuousCandidate ← GetFirst(CANDs);
2  LeastCost ← GetCost(Expand(BestContinuousCandidate),cUPM);
// Loop through the list of candidates; select the best one
   through a series of binary comparisons.
3  for currCandidate in CANDs do
       // Only proceed with the evaluation if the candidate may
          have a lower cost
4      if GetBasicCost(Expand(currCandidate),cUPM) < LeastCost then
           // Update the current best candidate and incumbent value
              if the candidate is better than the best candidate
              found so far
5          if GetCost(Expand(currCandidate),cUPM) < LeastCost then
6              BestContinuousCandidate ← currCandidate;
7              LeastCost ←
               GetCost(Expand(BestContinuousCandidate),cUPM);
8          end
9      end
10 end
11 return BestContinuousCandidate
```

Algorithm 10: DequeueBestContinuousCandidate

their lower and upper bounds. The sum of all costs is recorded as the cost of CR.

```
   input  : ContRlx, a candidate continuous relaxation
   input  : cUPM, the continuous user preference model
   output: Cost, the cost of CR according to cUPM

   // Initialize the cost value.
1  Cost ← 0;
   // Loop through all relaxed temporal constraint in ContRlx.
2  for RC in ContRlx do
       // Check the modification made to the lower and upper bounds
          of RC.
3      ΔUB_RC = RelaxedUB(RC)-UB(RC);
4      ΔLB_RC = RelaxedLB(RC)-LB(RC);
       // Compute the cost of the relaxed constraint.
5      Cost = Cost + a_LB^RC ΔLB_RC + b_LB^RC;
6      Cost = Cost + a_UB^RC ΔUB_RC + b_UB^RC;
7  end
8  return Cost
```

Algorithm 11: GetCost

To find the candidate with the lowest cost, DequeueBestContinuousCandidate has to compute the cost of each continuously relaxed temporal constraints in each candidate. To avoid redundant computation, we are using a Branch and Bound approach here with an incumbent value to prune candidates that cannot provide a lower cost, hence avoid the evaluation process of them. DequeueBestContinuousCandidate uses *LeastCost* as an incumbent value. Each time *LeastCost* is updated, all candidates whose **basic costs** are larger than the incumbent will be excluded from the cost evaluation. The basic cost of relaxing a temporal constraint is the lowest cost of a relaxation that relaxes it. If the basic cost of a candidate is larger than the incumbent, it cannot be a better candidate since its total cost must be larger than the incumbent value.

Next, DequeueBestContinuousCandidate evaluates one of the remaining candidates. If the remaining candidate costs less than the incumbent, the incumbent will be updated and used to prune more candidates. Otherwise this candidate will

be excluded from the evaluation.

4.4.2 Proving the Optimality of Continuous BCDR

Given an inconsistent OCSTN, $\mathcal{P}$, we claim that CONTINUOUS BCDR generates the most preferred minimal continuous relaxation to $\mathcal{P}$, given a semi-convex preference function over $\mathcal{P}$. We prove the optimality of CONTINUOUS BCDR in three steps:

- CONTINUOUSLYRESOLVECONFLICT generates the most preferred minimal **constituent relaxation** to a single minimal conflict.
- UPDATECONTINUOUSCANDIDATES and DEQUEUEBESTCONTINUOUSCANDIDATE generate the most preferred minimal **continuous relaxation candidate** to a set of minimal conflicts.
- CONTINUOUS BCDR generates the most **preferred minimal relaxation** to an inconsistent OCSTN.

The Optimality of CONTINUOUSLYRESOLVECONFLICT

Given a set of minimal conflicts, $MinCFLTs$, we would like to generate the most preferred minimal continuous relaxation that resolves all the conflicts using a 2-step approach: UPDATECONTINUOUSCANDIDATES and DEQUEUEBESTCONTINUOUSCANDIDATE. We make the assumption that the user preference functions over the relaxations to temporal constraints are semi-convex. Further, the cost of a relaxation increase linearly with Δ_{LB} and Δ_{UB}.

This can be framed as a linear optimization problem: we select the values of Δ_{LB} and Δ_{UB} to each temporal constraint so that all the constraints imposed by $MinCFLTs$ can be satisfied. The standard approach to the problem is a LP solver which computes the optimal relaxations to each temporal constraint that minimize the cost.

In this thesis, we take another approach that makes use of the property of semi-convex functions. Instead of using an optimization algorithm, we use a shortest path

algorithm (Function ContinuouslyResolveConflict) to generate the continuous relaxation that minimize the cost.

Theorem 10. *Given a semi-convex preference function, $\mathcal{UPM}$, and a minimal conflict, $MinCFLT$, the tightest continuous relaxation to a temporal constraint, TCR to TC, computed by Floyd-Warshall has the least cost among all continuous relaxations of TC against $MinCFLT$.*

Proof. [Proof by contradiction]

Assume that there is a continuous relaxation to $MinCFLT$, TCR', which relaxes constraint TC and has a lower cost than TCR.

1. If TCR' has a lower cost compared to TCR, given that all cost functions are semi-convex, the relaxation made by TCR' must have a smaller modification compared to TCR.

2. However, TCR indicates the tightest bounds to TC computed by APSP, which means that any bounds smaller than that in TCR will not resolve the minimal conflict.

Hence the assumption does not hold. TCR is the best continuous temporal relaxation. □

The shortest path algorithm provides the tightest bound of a relaxation that can resolve a conflict. Therefore, the relaxed temporal bounds in any other relaxation must be wider. Given that semi-convex functions prefers less modifications to the temporal bounds, the tightest bound computed by ContinuouslyResolveConflict is the most preferred one.

The Optimality of the Candidate Relaxations

Next, we prove that the *BestContinuousCandidate* generated by UpdateContinuousCandidates and DequeueBestContinuousCandidate is the most preferred candidate continuous relaxation to all known conflicts based on Lemma 4.

Lemma 4. *[Optimality of* UpdateContinuousCandidates*] Given a set of minimal conflicts, $MinCFLTs$, and a minimal candidate continuous relaxation CMR*

generated by UpdateContinuousCandidates *that relaxes a set of temporal constraints TCs, CMR is the most preferred candidate minimal relaxation among all continuous relaxations to $MinCFLTs$ that relaxes TCs, according to a semi-convex preference function UPM.*

Proof. [Proof by contradiction]

Assume that there exists another candidate minimal relaxation, CMR', to $MinCFLTs$ that also relaxes TCs but costs less than CMR'.

- Given that UPM is a semi-convex preference function, if MCR' costs less then MCR, then at least one of the relaxed temporal bounds in MCR' is narrower than that in MCR.

- However, all the temporal bounds in MCR are computed by Floyd-Warshall and are minimal in that they cannot be made narrower to resolve all conflicts in $MinCFLTs$.

  Therefore, the assumption does not hold and MCR is the most preferred minimal continuous relaxation that relaxes TCs.

□

Theorem 11. *[Optimality of BestContinuousCandidate] Given a new set of minimal conflicts, $MinCFLTs$ and a semi-convex preference function,* UpdateContinuousCandidates *generates all candidate minimal continuous relaxations $MCRs$ to $MinCFLTs$. There is a candidate MCR in $MCRs$ that is the* **most preferred** *candidate to $MinCFLTs$, and is selected by* DequeueBestContinuousCandidate.

Proof. [Proof by contradiction]

Assume that there is another minimal continuous relaxation, MCR', that resolves $MinCFLTs$ and costs less than MCR.

- First, since function DequeueBestContinuousCandidate is complete, $MCR' \notin MCRs$. Otherwise MCR' will be returned by the function.

- Second, function UpdateContinuousCandidates is complete in that it generates all minimal continuous relaxations to $MinCFLTs$. In other words, $MCRs$ covers all minimal combinations of continuous relaxations to the temporal constraints in $MinCFLTs$.

- Therefore, even though $MCR' \notin MCRs$, it must relax the same set of temporal constraints, TCs, with one of the candidate MCR_k in $MCRs$.

- However, by Lemma 4, MCR_k has the lowest cost compared to all other continuous relaxations that relaxes TCs. Hence the cost of MCR' cannot be lower than MCR_k and then hence MCR, which is the **most preferred** candidate in the set $MCRs$.

  The assumption does not hold and MCR is the candidate minimal continuous relaxation that can resolve $MinCFLTs$.

□

Therefore, we have proven that the candidates generated during the enumeration process are always optimal in terms of the known conflicts.

The Optimality of Continuous BCDR

Finally, we show that Continuous BCDR generates the optimal minimal continuous relaxation to an inconsistent OCSTN, given a semi-convex preference model.

Theorem 12. *[Optimality of* Continuous BCDR*] Given an inconsistent OCSTN, $\mathcal{P}$, and a continuous preference model, UPM, over $\mathcal{P}$, the first continuous relaxation generated by* Continuous BCDR *is the most preferred minimal continuous relaxation, CMR, to $\mathcal{P}$.*

Proof. There are two possible cases.

First, if all conflicts in $\mathcal{P}$, $MinCFLTs$, have been detected when CMR is returned, then CMR is the most preferred continuous relaxation according to Theorem 11.

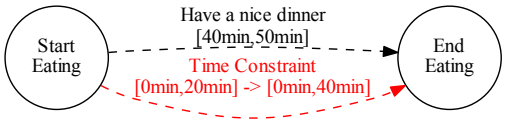
Second, if only a subset of all conflicts, $MinCFLTs' \subset MinCFLTs$, has been detected when CMR is returned, then CMR is the best candidate to $MinCFLTs'$, according to Theorem 11.

Further, if there is another minimal continuous relaxation, CMR', that resolves all conflicts in $MinCFLTs$, its cost must be larger than CMR. The reason is that CMR' also resolves $MinCFLTs'$, to which CMR is the best relaxation.

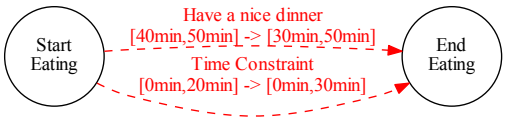
Therefore, CONTINUOUS BCDR generates the best minimal continuous relaxation to an inconsistent OCSTN.

□

Note that the best **minimal** continuous relaxation to an OCSTN may not be the continuous relaxation with the lowest cost. In other words, the best minimal continuous relaxation may cost more than the best continuous relaxation due to its requirements on minimality. For example, (Figure 4-12) shows two continuous relaxations, one minimal (Figure 4-12(a)) and one non-minimal (Figure 4-12(b)).



(a) A continuous temporal relaxation set of one relaxation



(b) A continuous temporal relaxation set of two relaxations

Figure 4-12: Examples of continuous temporal relaxations

Assume that the cost functions over constraints *Have a nice dinner* and *Time Constraints* are (Figure 4-13). Relaxing *Time Constraint* from [0min,20min] to [0min,40min] is the best minimal continuous relaxation that costs 100. However, relaxing both *Time Constraint* and *Have a nice dinner* by ten minutes costs only 90. This case demonstrates that sometimes slightly relaxing two constraints may cost less than relaxing

one constraint only, and the best minimal continuous relaxation may not be the best continuous relaxation.

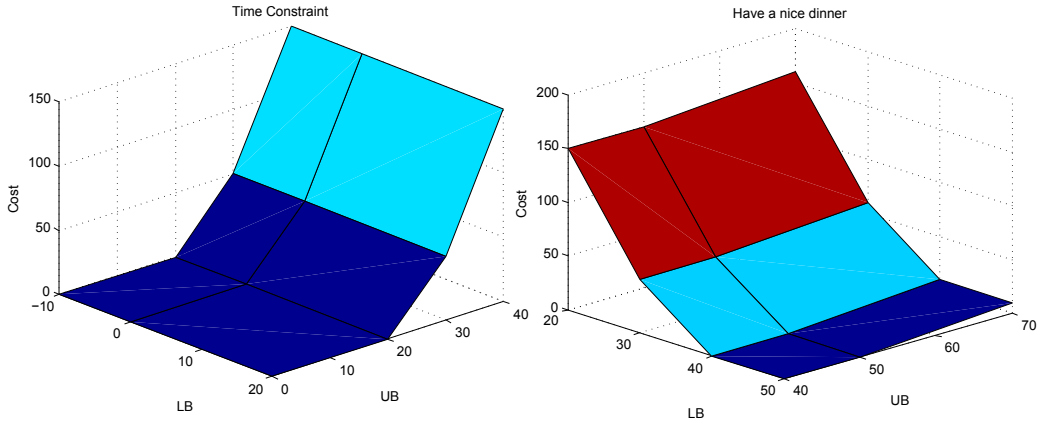


Figure 4-13: Continuous preference functions over the constraints

As a result, the minimality of continuous relaxations may prevent CONTINUOUS BCDR from generating the most preferred relaxation to an inconsistent temporal problem. On the other hand, if we drops the requirement on minimality, there will be infinite number of continuous relaxations to an OCSTN and they are impossible to be enumerated. For example, the second best continuous relaxation following (Figure 4-12(b)) may be relaxing *Time Constraint* by 11 minutes and *Have a nice dinner* by 9 minutes. The third best may relax the constraints by 12 and 8 minutes. Without the requirements on the minimality, the collaborative diagnosis process may become extremely inefficient due to the endless 'next best' resolutions. Hence we sacrifice the global optimality of the results generated for a more compact and efficient interaction between the user and Uhura.

In summary, we presented a new approach that generates the most preferred candidate continuous relaxation to inconsistent temporal problems. This is a 3-step method that first resolves each individual conflict using the minimal relaxation, then generates the most preferred relaxation candidate to all known conflicts during the enumeration and finally produce the best minimal continuous relaxation when enumeration terminates. In addition, we described an innovative method that compute the optimal relaxations efficiently: using a semi-convex preference function, the re-

laxation generated by shortest path algorithms has the lowest cost. Finally, given an inconsistent OCSTN and a semi-convex preference model, we proved that the Continuous BCDR algorithm enumerates minimal continuous relaxations in best-first order.

4.5 Chapter Summary

In this chapter we presented the concept and design of the Continuous BCDR algorithm used in Uhura. It is an innovative approach to the best-first enumeration of minimal continuous relaxations to inconsistent OCSTNs. Instead of suspending constraints, continuous relaxations preserve all the temporal constraints in the problem and resolves conflicts by minimally relaxing the temporal bounds of constraints. It addresses the third requirement of collaborative plan diagnosis: small perturbation. Compared to the discrete BCDR algorithm presented in Chapter 3, Continuous BCDR preserves the elements in the input problem to the maximum. It not only minimizes the constraints that are relaxed, but also minimizes the adjustments made to relaxed constraints.

Continuous BCDR is the first method that can generate continuous temporal relaxations to inconsistent OCSTNs. With the use of continuous preference models, Continuous BCDR generates the most preferred minimal continuous relaxation to an inconsistent OCSTN. The property of a semi-convex preference function guarantees that for each minimal conflict, the tightest relaxation of one simple temporal constraint is in fact the optimal relaxation. Continuous BCDR implements this property to compute minimal continuous relaxations efficiently using a shortest path algorithm. Similar to discrete BCDR, we implement Continuous BCDR using incremental set covering method, similar to CD-A* and DAA [10, 37, 4]. It makes Continuous BCDR preserve the anytime capability of discrete BCDR and provide quick responses to the users queries in real world applications.

Chapter 5

Experimental Results

Uhura has been incorporated within a model-based executive called Kirk [22] and a dialogue manager system [38, 36] in order to support collaborative diagnosis of over-constrained temporal plans. Kirk was developed as a model based plan executive that can generate threads of execution through the TPNs that are temporally consistent, and execute the partially ordered plan. The addition of Uhura enables Kirk to work with over-constrained TPNs: if no consistent thread of execution is found in a TPN, Kirk will call Uhura to initiate the collaborative diagnosis process and engage the user to resolve the conflicts.

In this chapter, we present the experiment results of Uhura on different test cases constructed based on the personal transportation scenario. Section 5.1 presents the experiment setup and the results of Discrete BCDR on discrete relaxations. In Section 5.2 we present the results of Continuous BCDR on continuous relaxation problems.

5.1 Generating Discrete Relaxations

In this section, we evaluate the effectiveness of BCDR on the basis of two criteria. First, we test the scalability of BCDR by benchmarking it with structured inconsistent OCSTNs with randomly selected parameters. The run-time performance is compared with AllRelaxation and Dualize & Advance. AllRelaxation is the baseline algorithm which enumerates all possible temporal relaxations using a brute force strategy. It demonstrates the cost of exploring the complete search and result space of a relaxation problem. Dualize & Advance uses conflict-directed techniques to enumerate all discrete minimal relaxations. It demonstrates the effectiveness of using minimal relaxations in terms of search space and size of results, and the problem of not considering user preferences.

Second, we evaluate the run-time performance of BCDR against problems with various levels of difficulty. The difficulty of a relaxation problem is measured by the percentage of episodes that need to be relaxed in order to restore the consistency of over-constrained OCSTNs. Section 5.1.1 describes the design of our experiments. The results of two experiments are presented in Section 5.1.2 and 5.1.3.

5.1.1 Experiment Setup

We ran two sets of experiments in the evaluation. The first one tests the scalability of the three algorithms using OCSTNs with various numbers of temporal constraints. The second one tests the performance of BCDR against OCSTNs of various difficulties.

Tests on Scalability

The complexity of a relaxation problem highly depends on the structure of the OCSTN: given a fixed number of temporal constraints, the number of minimal temporal relaxations increases linearly against the number of choices, and exponentially against the number of temporal constraints activated by each decision. The test cases are generated in a semi-randomized manner. We define seven classes of OCSTNs: the

OCSTNs in each class has the same number of temporal constraints, and the number varies from 20 to 300. For a problem with 300 temporal constraints, the number of possible temporal relaxations can be as large as 10^{45}. This is the biggest test setting we have seen in literature, and should be able to push all the algorithms in the experiment to the limit.

Within each class of problem, a set of *problem configurations* is defined. A *problem configuration* is a set of parameters that defines a certain problem structure. For example, a configuration of 20 constraints, 2 decisions and 10 constraints per decision defines a type of problem similar to (Figure 5-1). We cover most possible problem structures using configurations with different numbers of choices. For example, for 20-constraint class test cases, the number of choices can be any integers between 2 and 10. The number of temporal constraints per decision is adjusted accordingly so that the total number of constraints is around 20.

Within each configuration, ten different problems are randomly generated by varying the temporal bounds of temporal constraints and cost functions. For example, within the 20-constraint class, ten different configurations are available, ranging from 2 decisions and 10 constraints/decision (Figure 5-1) to 10 decisions and 2 constraints/decision (Figure 5-2). Note that we use dummy constraints with temporal bounds of [0,0] in the graph to separate the effective constraints. The dummy constraints have no effect on the result and are not counted towards the total number of constraints in each problem configuration. Overall, we created 1460 test cases using the following parameters:

Number of temporal constraints: 20,50,100,150,200,250,300.

Number of decision events: between 2 and 150.

Number of temporal constraints in each decision: between 2 and 150.

Number of Events: Equal to the number of constraints.

Number of preference levels: Equal to the number of constraints.

Constraint Domain: between 0 and 100

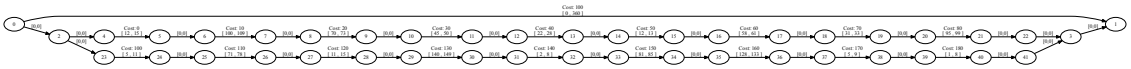


Figure 5-1: 20-constraint test case: 2 decisions with 10 constraints

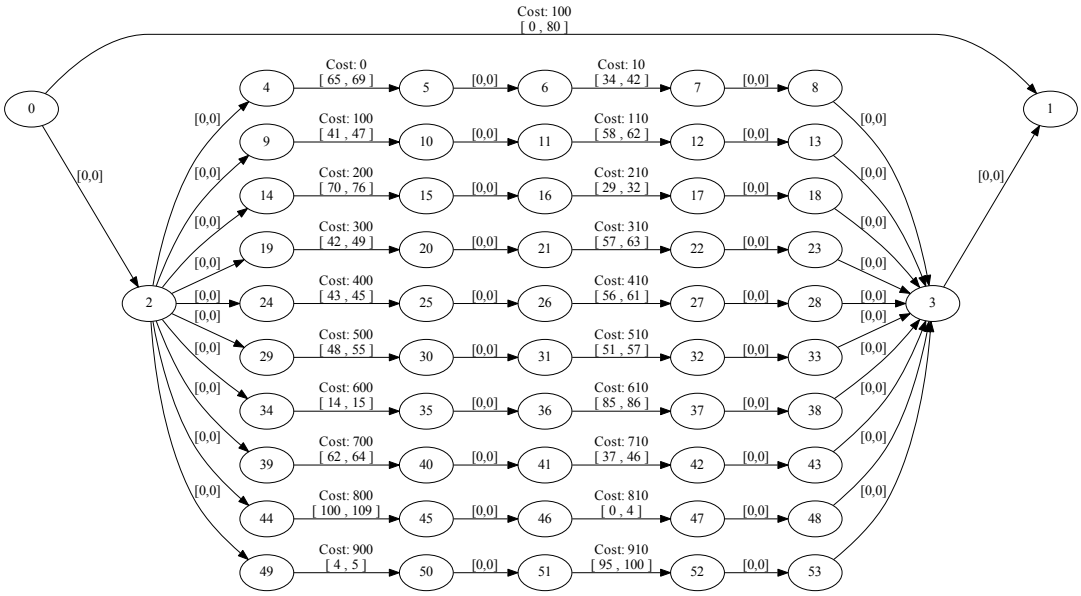


Figure 5-2: 20-constraint test case: 10 decisions with 2 constraints

There is a special constraint called *controller* constraint that governs all disjunctions in each OCSTN, such as the temporal constraint between the start and end events in (Figure 5-1). The *controller* constraints are used to adjust the consistency of the OCSTN. Within each disjunction, the temporal bounds of each temporal constraints are randomly selected between 0 and 100. In order to make all test cases inconsistent, we adjust the upper bounds of the *controller* constraints, $UB_{controller}$, such that the lower bounds of all temporal constraints in each decision sum up to 80% of $UB_{controller}$.

For example, in (Figure 5-2), the sum of constraint '4-5 [65,69]' and '6-7 [34,42]' is '[99,111]'. Compared to the controller constraint, '0-1 [0,80]', the lower bound of the sum exceeds by 20%. The use of controller constraints guarantees that all test cases are inconsistent, and in general one-fifth of the temporal constraints in each disjunction have to be relaxed in order to make the OCSTN consistent. Table 5.1

Constraint number	Number of configurations	Number of random problems
20	10	100
50	11	110
100	17	170
150	21	210
200	25	250
250	30	300
300	32	320

Table 5.1: Specification of benchmark Optimal Conditional Simple Temporal Networks

summarizes the specs of test cases in each class.

All test cases in this experiment are structured following this guidelines. Note that an unstructured random temporal problem generator is described in literature [33]. Given a fixed number of events, N_e, and the ratio between the number of temporal constraints (N_c) and N_e, $\mathcal{R} = N_c/N_e$, the generator creates temporal problems by creating temporal constraints of random durations and disjunctions between events. It is shown in [27] that the percentage of inconsistent problems created by this generator for $\mathcal{R} = 2, 3, 4, 5, 6, 7$ were (0%,0%,0%,12%,72%,94%), respectively. In other words, the generator can hardly create inconsistent problems when N_c is less than four times of N_e. However, in our problem settings of the Personal Transportation system scenario, the value of $\mathcal{R}$ is usually lower than 2. The inconsistent OCSTNs are usually the result of one or several user defined constraints that are too tight compared to the restrictions imposed by the environment. The generator in [33] can hardly provide any inconsistent OCSTNs in our settings. Therefore, we choose to take the structured approach to generate test cases for BCDR.

Tests on Difficult Problems

The second group of tests focuses on the effect of difficult problems on the run-time performance of BCDR. To evaluate the difficulty of a OCSTN, we define a parameter **Over-constrained Level** of the OCSTN as:

Definition 43. *The **Over-constrained Level**, $\mathcal{OC}$, of a OCSTN, P, is $\mathcal{OC} = N_{rtc}/N_{tc}$, where*

- N_{rtc} *is the average number of relaxed temporal constraints in the minimal relaxations to* P.

- N_{tc} *is the number of temporal constraints in each decision of* P.

This parameter is used to evaluate the difficulty of a temporal relaxation problem, since it affects the number of possible minimal relaxations to the inconsistent OCSTN. For example, given a 10% over-constrained OCSTN in the 100-constraint class, the number of minimal relaxations to it is around 10^6. For a 50% over-constrained OCSTN, the number may rise to 10^{14}.

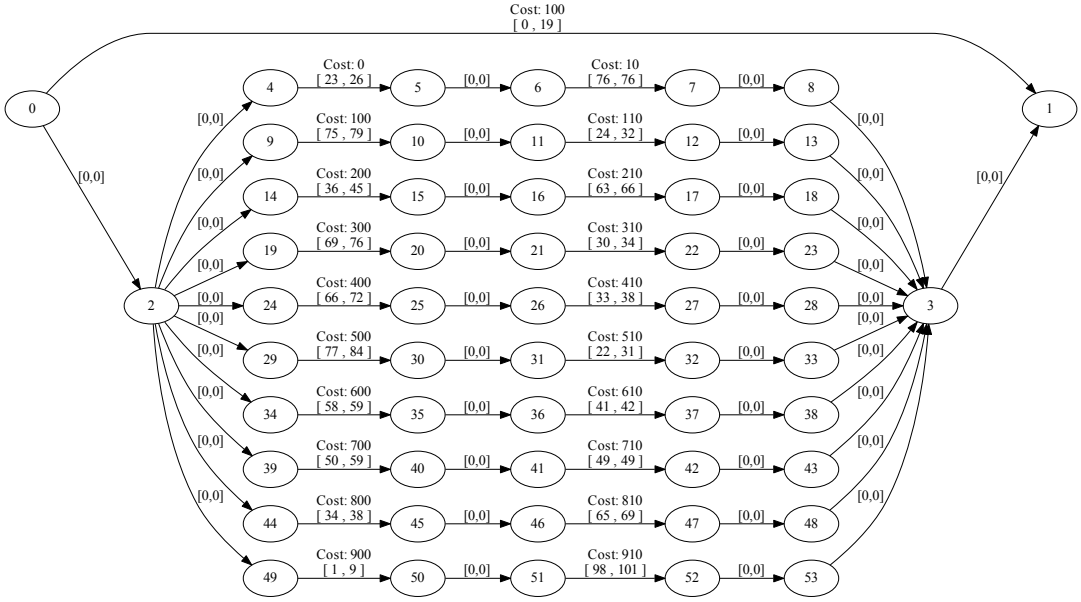


Figure 5-3: 20-constraint test case: 80% over-constrained

In this group of test cases, this parameter is controlled by the upper bounds of the *controller* temporal constraint: the over-constrained level is the ratio between the sum of the lower bounds of all temporal constraints in one choice, Sum_{LB}, and the upper bound of the controller temporal constraint, UB_{ctc}. If $Sum_{LB}/UB_{ctc} = \mathcal{OC}$, then the problem is said to be $100\mathcal{OC}$% over-constrained.

For example, (Figure 5-1) is a 20% over-constrained problem, while (Figure 5-3) is a 80% over-constrained problem. In this group of experiments, we define three

Constraint #	Over-constrained Levels	# of configurations	# of random problems
20	20,30,40,50,60,70,80	10	700
50	20,30,40,50,60,70,80	11	770
100	20,30,40,50,60,70,80	17	1190

Table 5.2: Specification of over-constrained level test cases

classes of test cases: 20 constraints, 50 constraints and 100 constraints. Within each class, the test cases varies in the structures and over-constrained levels. We created 2660 test OCSTNs using parameters in (Table 5.2).

All algorithms are benchmarked based on the number of temporal consistency checks called during the relaxation process to eliminate the variation caused by computers. AllRelaxations is set to generate all full temporal relaxations. Dualize & Advance is set to generate all minimal temporal relaxations. BCDR is set to generate the 10 most preferred minimal temporal relaxations, if available.

5.1.2 Analysis of Scalability

All tests are completed on a Core i7 computer with 12GB RAM. The result of each class is an average number of consistency checks, which is averaged from the numbers of all test cases in that class. The maximum number of consistency checks allowed on each test run is 10^4. Each dot in the graph represents an individual test run, and the line shows the average number in each class of test cases.

As shown in (Figure 5-4), AllRelaxations (the brute force algorithm) performs the largest number of consistency checks, which times out on the 50-constraint problem. The reason for its poor performance is that it tries to enumerate and test all candidate temporal relaxations. The number of candidate temporal relaxations to a 20-constraint OCSTN is around 10^3, and it quickly rises to 10^{15} for an OCSTN in the 100-constraint class.

Compared to AllRelaxation, only enumerating minimal relaxations using Dualize & Advance significantly reduces the run time: inconsistent OCSTNs with less than 50 constraints are solved in less than 1000 consistency checks, which is roughly equal to 1 second of computation time on a regular desktop computer. The improvement

in performance is due to two factors. First, DAA uses conflicts to guide the search away from infeasible relaxations, and towards feasible relaxations. Second, it avoids generating non minimal relaxations. Both factors improve performance, and the second factor also reduces the number of options presented. The candidate **minimal** relaxations to a 100-constraint OCSTN is around 10^{10}, which is 100,000 less than that of AllRelaxations (10^{15}). Note that the Dualize & Advance algorithm is implemented with the general minimal conflict extract method, which does not make use of the negative loops in temporal problems to generate conflicts (such as [19]).

The computation time of Dualize & Advance algorithm is still impractical in most real-world scenarios. The number of consistency checks required exceeds 10^5 when the number of temporal constraints in the OCSTN is larger than 100. As stated in Chapter 1, to enable collaborative diagnosis, the autonomous decision system should respond quickly to inconsistent temporal problems. The waiting time for the user should not exceed 1 second, which is roughly equal to 10^3 consistency checks.

Compared to AllRelaxation and Dualize & Advance, Uhura (using the BCDR algorithm) with an improved minimal conflict extraction algorithm and user preference models achieves 10^2 higher run time performance. In this experiment, BCDR only generates the ten most preferred minimal relaxations. It runs significantly faster on all problems than AllRelaxation. Compared to Dualize & Advance, BCDR reduces the number of consistency checks by more than two orders of magnitude. It avoids the minimization process of DAA that iterates through every temporal constraints in an inconsistent candidate. This saves nearly 90% of the consistency checks. In addition, the number of minimal temporal relaxations that needs to be enumerated is nearly 100 times less than that of DAA due to the use of preference models. BCDR stops enumeration when the tenth relaxation is generated.

Finally, (Figure 5-5) shows the run-time performance of BCDR against inconsistent OCSTNs with different numbers of choices. Each line in the graph represents the result with regarding to a problem with a certain number of choices. Given a fixed number of constraints, the number of temporal constraints per choice decreases when the number of choices increases.

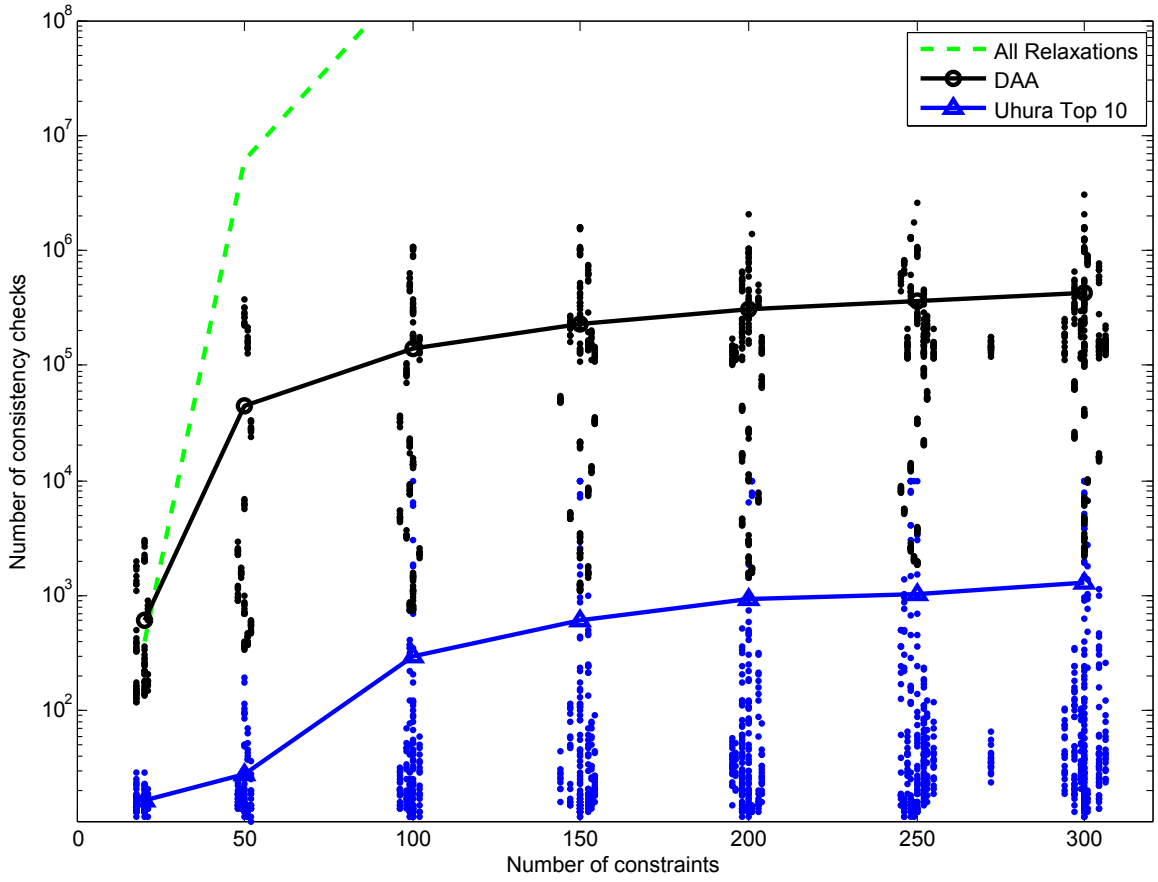


Figure 5-4: Runtime on randomly generated temporal problems with different numbers of constraints

As can be seen in the figure, BCDR's performance improves when the number of choices increases (or, the number of temporal constraints per choice decreases), regardless of the total number of episodes in the problem. As stated in the previous section, the complexity of a relaxation problem increases linearly against the number of choices, and exponentially against the number of temporal constraints in each disjunction. Therefore, given a fixed number of temporal constraints, an OCSTN with a smaller number of choices is generally more complex and harder to resolve than an OCSTN with more choices.

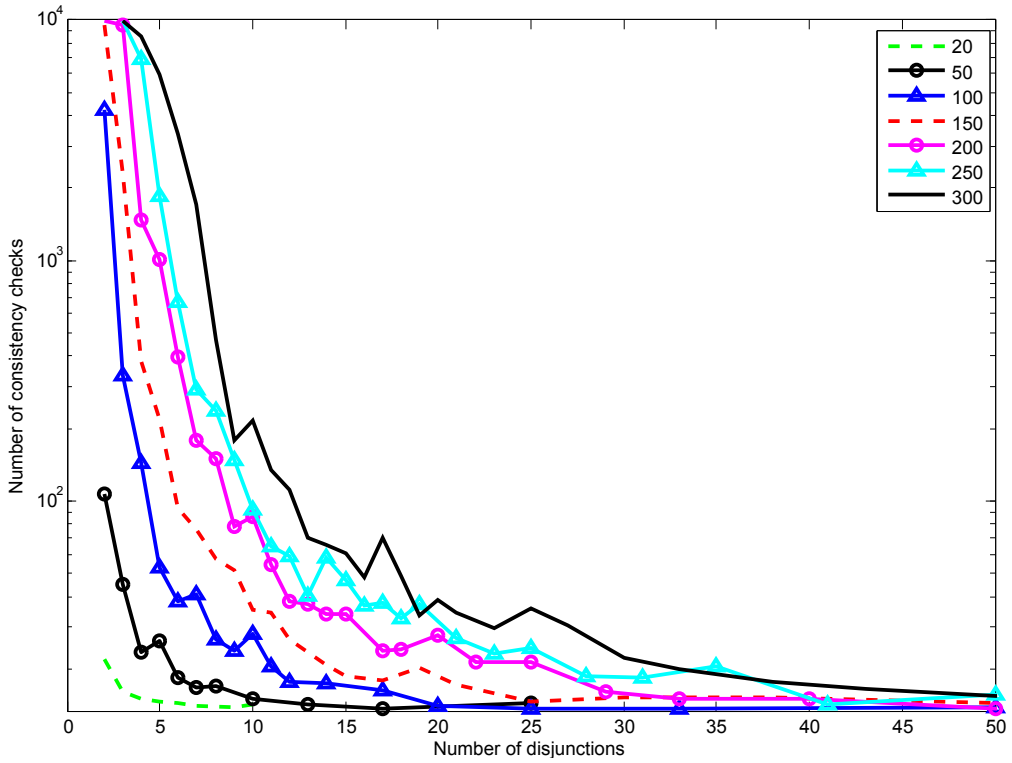


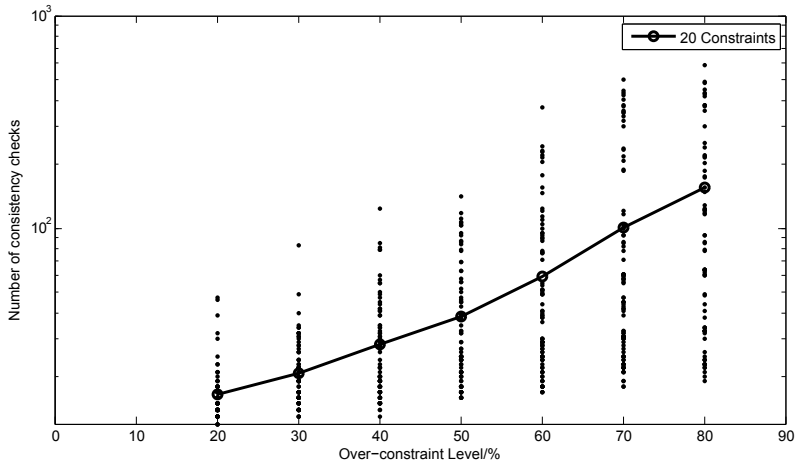
Figure 5-5: Runtime of Uhura (using BCDR) on temporal problems with different numbers of choices

5.1.3 Analysis of Performance on Difficult Problems

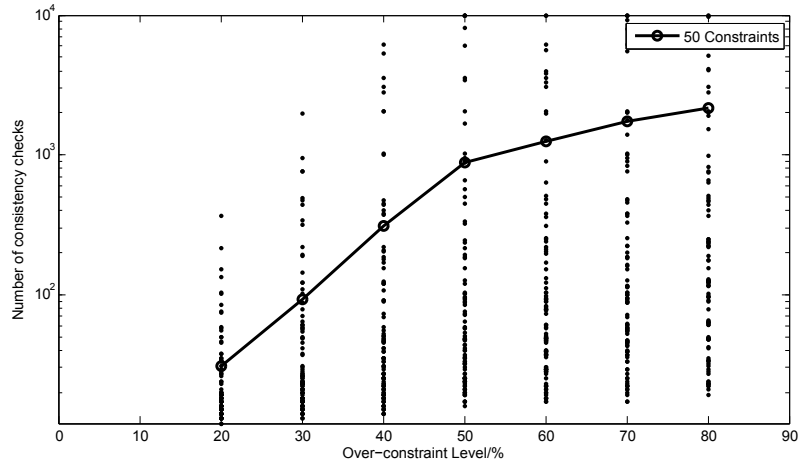
(Figure 5-6) shows the runtime of Uhura (using BCDR) on inconsistent OCSTNs with different difficulties (over-constrained levels). The dots in the graph represent the result of each individual test case and the solid lines represent the average across all test cases in each category. The vertical axis represents the number of consistency checks for each test case, and the horizontal axis represents the over-constrained levels. Recall that the over-constrained level of an over-constrained OCSTN indicates how many simple temporal constraints need to be relaxed on average to restore the temporal consistency.

It can be seen from the graph that the number of consistency checks required by BCDR increases with the over-constrained levels. As stated in the previous section, the number of minimal temporal relaxations to a OCSTN may increase if its over-constrained level increases, since a harder problem usually has more conflicts and requires more consistency checks to resolve. In addition, for best-first enumerations,

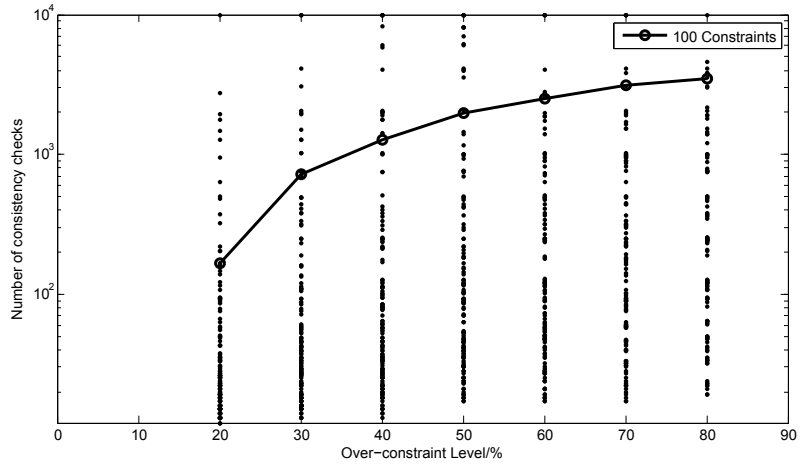
problems with higher over-constrained levels usually require many more computations than slightly over-constrained problems. This is the result of our assumption that the user prefers relaxations that relax fewer temporal constraints. Based on this assumption, BCDR will always test candidates that relax less constraints first, hence steer the enumeration away from the correct resolutions.



(a) 20 constraints



(b) 50 constraints



(c) 100 constraints

Figure 5-6: Runtime of Uhura (using BCDR) on temporal problems with different over-constrained levels

5.2 Generating Continuous Relaxations

In this section, we evaluate the performance of CONTINUOUS BCDR on continuous relaxation problems. We continue to use the structured inconsistent OCSTNs presented in Section 5.1 to benchmark CONTINUOUS BCDR. Recall that the number of simple temporal constraints, choices and temporal bounds vary in each test case. The test results are compared to DISCRETE BCDR, which generates discrete relaxations and is supposed to be faster due to the direct suspension of constraints. We use the runtime of the algorithms in milliseconds to compare their performance. All experiments are done on a Core i7 computer.

5.2.1 Analysis of Scalability

As we presented in Chapter 4, the generation of continuous relaxations can be viewed as generating discrete relaxations plus temporal bounds tightening. Given that we use the Floyd-Warshall algorithm to compute the tightest constraint bounds of each candidate, which is a polynomial algorithm in terms of the events, the additional runtime required by continuous relaxations should be polynomial, too.

We use the scalability test cases presented in Section 5.1, which are constructed based on the PTS scenario. Recall that there are seven classes of tests, each representing a number of constraints ranging from 20 to 300. Within each class, we define several subclasses of tests with different numbers of choices. For a given structure, we randomly generate five test cases by varying the temporal bounds of constraints. Therefore, we have in total 1600 test cases that cover most daily trip scenarios of the Personal Transportation System.

There is a major difference between this experiment and the one in Section 5.1: we use run time instead of consistency checks to benchmark the algorithm. The numbers of consistency checks are usually identical in both discrete and continuous approaches, since the CONTINUOUS BCDR is only different from DISCRETE BCDR in that it has an additional temporal bounds tightening process for constituent relaxations. We would like to know the additional computation required for doing continuous

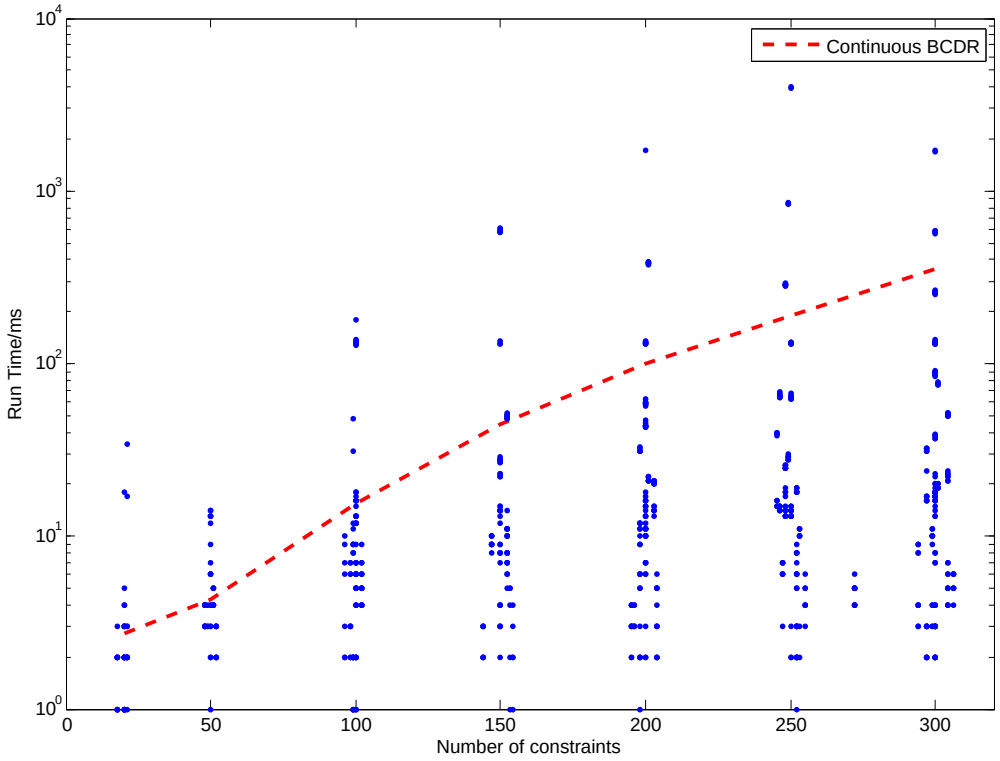


Figure 5-7: Runtime of continuous BCDR on relaxation tests

relaxation compared to discrete relaxation. The run time performance of Discrete BCDR and Continuous BCDR is presented in (Figure 5-8 and 5-7).

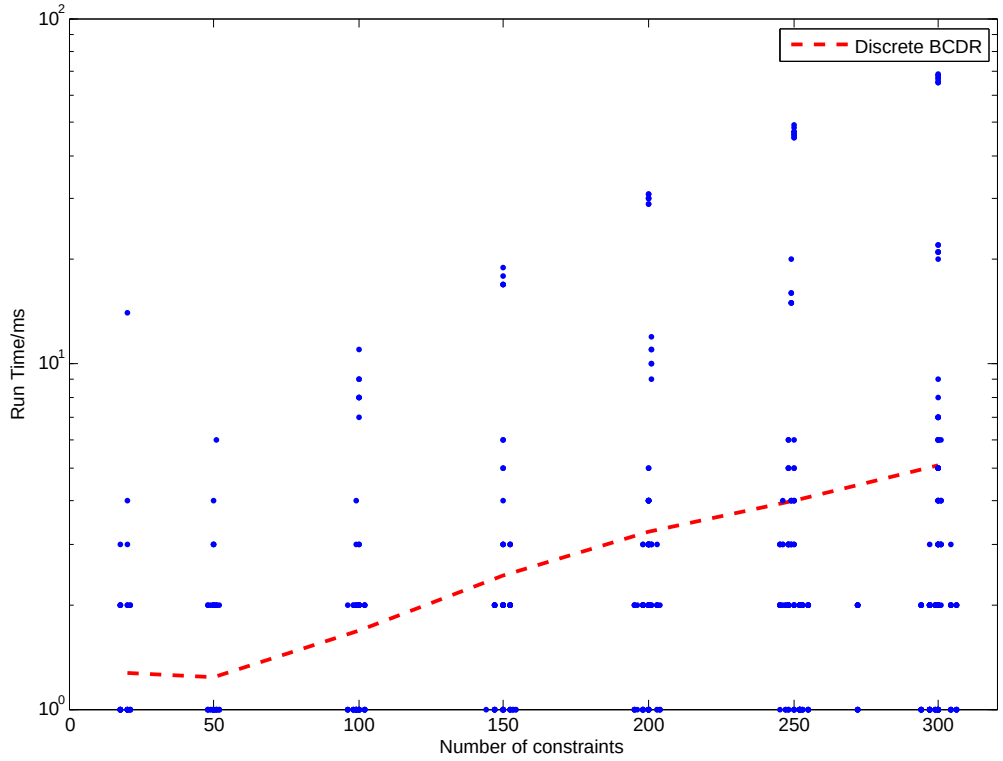


Figure 5-8: Runtime of discrete BCDR on relaxation tests

In these experiments, we constrain both algorithms to generate the first ten relaxations to the inconsistent test cases. The blue dots in the graph represent the run

time of each individual experiment, and the red lines show the average run time of each class of test cases. The horizontal axis represents the number of constraints in the problem, and the vertical axis represents the run time of the algorithm in millisecond. As can be seen from the graph, the run time of CONTINUOUS BCDR increases exponentially against the number of constraints in the test case. This coincides with the result we get from DISCRETE BCDR: the run time is dominated by the number of consistency checks, and the number of consistency checks required is exponential in terms of the constraints.

The slope of the continuous relaxation curve is larger than the curve of discrete relaxations. In other words, it takes more time to generate ten continuous relaxations than discrete relaxations. As we presented in Section 5.2, there is an additional step in CONTINUOUS BCDR that computes the minimal relaxed temporal bounds of suspended constraints. It uses a polynomial algorithm, Floyd-Warshall, that runs in $O(n^3)$, where n is the number of events in the conflicts. On the other hand, DISCRETE BCDR resolves conflicts through constraint suspension, which saves the extra time spent on computing continuous constituent relaxation when a conflict is detected.

The time difference between (Figure 5-8) and (Figure 5-7) is increasing with regards to the number of constraints due to the increasing size of conflicts: the additional time required to compute the temporal bounds is longer on large conflicts. It starts from 5 ms on 20-constraint problems to nearly 500 ms on 300-constraint problems. If the user is sensitive to the runtime and needs a quick response, generating the discrete relaxations using DISCRETE BCDR would be a better approach, if he is not critical about the quality of the result. On the other hand, if the user wants to minimize the modification to the plan requirements, he may spend 2 to 5 times more time and get continuous relaxations using CONTINUOUS BCDR.

5.3 Chapter Summary

In this chapter, we present the benchmark results of Uhura. There are two groups of tests performed on discrete and continuous relaxation generation. We tested Discrete BCDR and Continuous BCDR algorithms separately, then compared their run time performance. Within each group of tests, we analyzed the scalability of the algorithms, as well as their performance on problems of different difficulty.

The experiment results show that BCDR achieves significant improvement in run-time performance on large and difficult temporal problems compared to previous approaches. The conflict-directed technique efficiently prunes the search space in enumeration process. In addition, the inference based conflict extraction method and best-first enumeration strategy make BCDR nearly two orders of magnitude faster than Dualize & Advance. In addition, the use of minimal relaxations reduced the result spaces compared to the baseline algorithms, AllRelaxation, by several orders of magnitude.

Finally, we compared the discrete and continuous version of BCDR in terms of run time performance. As presented in Chapter 4, Continuous BCDR requires an additional polynomial time process of temporal bounds tightening. The experiment results verified our hypothesis on the difference between their performance: Continuous BCDR is one to two orders of magnitude slower than Discrete BCDR due to the additional step. As a result, we recommend using Discrete BCDR if the user is more sensitive to quick response. If the user asks for a higher quality results, then Continuous BCDR would be a better fit.

Chapter 6

Summary and Future Work

This chapter begins by presenting several ideas for future extensions of the research presented in this thesis. We then conclude this thesis and discuss our contributions in Section 6.2.

6.1 Future Work

In this section, we present ideas for extending the capabilities of Uhura and its BCDR algorithm and new applications as future work. This section is divided into two parts. First, we describe the open questions in the current approach to discrete and continuous relaxations that have not been addressed. Second, we present possible extensions to Uhura that provide new capabilities and applications.

6.1.1 Open Questions Within the Current Approach

Repairing Dependencies of Relaxed Constraints

Usually, in a temporal plan, the presence of some constraints are conditioned on the presence of other constraints. For example, the constraint on John's dinner duration and the driving time to restaurants are only valid if he decides to go out for dinner. If he chooses not to have dinner on his way back home, none of these constraints still makes sense. As a result, there is dependency between constraints that are relaxed in that the relaxation of some constraints should imply the relaxation of other constraints.

In our current approach, this dependency is not considered during relaxation: Uhura only checks the temporal consistency of conditional constraints guarded by choices. To add the dependencies between constraints, we need a new encoding of conditional constraints as well as a dependency checking method: each time a constraint c_k is relaxed, Uhura should be aware of the other constraints C_{dk} that depend on the existence of c_k, and generate corresponding relaxations to them.

A possible approach to addressing this issue is to improve the existing encoding of conditional constraints. Recall that Uhura is capable of relaxing OCSTNs, in which the activation of temporal constraints may depend on the choice made to decision events. By including the choices in both conflict and relaxation representations, Uhura can guarantee the consistency between choices and activated temporal constraints. We can add the dependencies between constraints as an additional requirement for activating conditional constraints. For example, if the activation of constraint c_k

requires both choice $\mathcal{D} \leftarrow k$ and the activation of constraint c_j, then Uhura will deactivate c_k whenever c_j is relaxed. In other words, c_k can only be preserved if c_j is activated and preserved.

Generating Good English Explanations

In this thesis, we have discussed the challenges and approaches of generating temporal relaxations to inconsistent problems. However, Uhura is designed as an interface between the user and a temporal planner, and is supposed to communicate the alternatives to the users using natural languages. As a passenger of PTS, John will be expecting the explanation from Uhura in the form of plain English, not a statement consisting of temporal constraints and events. To achieve this, we have to address another challenge of generating good English explanations back to the user.

An explanation is good in three aspects. First, as stated before, the explanation must be presented using plain English. Uhura is developed as an automatic taxi driver system. The target users of Uhura are general public without any background in constraint programming and artificial intelligence. It is unrealistic to expect a non-AI expert to understand temporal constraints, conflicts and minimal relaxations. Therefore, the explanations should avoid this jargon and use descriptive expression to describe the problems and alternative plans.

Second, the expression must be at the appropriate level of abstraction. The temporal plan can be as large as thousands of constraints, and simply presenting it to the user may cost a lot of time. To make the explanation succinct, Uhura has to identify the key elements of a relaxation that must be communicated to the user, the supporting reasons for these relaxations and the elements that remains unchanged or not related. While presenting relaxations, Uhura should only communicate the key modifications in the alternative plans, provide reasons when asked and hide the unrelated details from the user. A key enabler to this capability is a shared knowledge model between Uhura and the user, so that Uhura can evaluate each piece of information and avoid telling the user what he or she already knows, or missing supporting evidences that may cause confusion.

Finally, generating an explanation is not all about language. In most situations, it would be more efficient if the explanation can be presented using both graphical interfaces and verbal communications. For example, to describe a flight path from Boston to Detroit, it may take Uhura several minutes to speak out each navigation and check points on the route. On the other hand, the passenger may get the message in ten seconds if a route map is presented. This is a challenge similar to the second one, for which Uhura has to choose the most efficient way to present each part of an alternative plan.

6.1.2 New Capabilities and Applications

Relaxing Constraints on States

In this thesis, we discussed the resolution of over-constrained temporal problems using continuous and discrete temporal relaxation. Uhura adjusts the temporal bounds of constraints in an inconsistent OCSTN in order to restore its consistency, which is equivalent to relaxing the temporal goals in the QSP. However, another way is to modify the user's state goals in the QSP in order to enable a consistent plan to be generated. For example, if all of the sandwich restaurants are too far away from John's office to meet his temporal goal, it is natural to ask him if he can relax his requirements of the restaurant type. say from sandwich restaurant to any fast-food restaurant. Therefore, more alternative restaurants will be available for John, and some of them may be closer to his route home. If again no plan to these restaurants satisfies the temporal goal, we can further relax the state goal from fast-food restaurant to any restaurants until a consistent plan is generated.

This feature requires a new capability that is not available in Uhura at the moment. Uhura have to identify the type of the state goals and construct the relaxation type hierarchy for them. Then relax the state goals following the type hierarchy, from lower levels to upper levels. For example, Cosi is a sandwich restaurant, and a sandwich restaurant is a type of fast-food restaurant. If no feasible plan is found that achieves all the user's goals, Uhura may relax this state goal from Cosi to any

sandwich restaurant and then to any fast-food restaurant. This type of relaxation introduces more options into the planning problem. Hence it increases the chance of generating a feasible plan: there may be one restaurant that is close enough to John's route home that will save him a lot of time on driving.

To enable this capability, we have to solve two problems: when to relax and how to relax a state goal, given an inconsistent plan. To address the first problem, we need to introduce a new way to resolve a known conflict: modifying its state constraints. Currently we are using two ways to resolve conflicts detected in an inconsistent OCSTN: relaxing some temporal constraints, leaving the state constraints unchanged; or changing the choice to deactivate some state and temporal constraints in the conflict. The later method may be modified to provide this capability. For each state constraint in the plan, we may encode an alternative that represents its relaxation using a decision event. Therefore, when a minimal conflict is detected, BCDR can generate a constituent relaxation that asks for a relaxation over the state constraint.

The second problem, how to relax a state goal, may be addressed by introducing a structure that can store type information and answer queries about relaxations. A similar method has been used in WordNet [26], a lexical database of English that groups words into sets of synonyms and are organized into hierarchies. It can be used to answer queries like the classes of a word, such as dog $\rightarrow$ mammal $\rightarrow$ animal.

Interactive Plan Diagnosis

In Chapter 3, we presents Uhura using a simplified user-robot interaction model, in which the only job of John is to make decisions while the robot sends back a proposed temporal relaxation. If John accepts the proposal, Uhura will terminate the enumeration and implement the relaxations. If John rejects, Uhura continues to look for the next best relaxations to John's inconsistent problem.

However, in real world scenarios, human and robots are usually working together. They may not have the complete set of information of the task, and need to provide explanations and feedbacks to each other in order to generate a shared plan. For

example, Uhura may ask John if he can postpone the arrival time, while John may ask Uhura if he can drop by a grocery store and pick up snacks for his party.

To solve the over-constrained situation, both human and robot have to make contribution: the operator has to tell the robot his or her preferences over the options, while the robot need to communicate the environment constraints and evaluate the feasibility of all goals proposed by the operator. This capability will require a different collaboration model, in which the human and robots work like peer-to-peer instead of leader-assistant.

To satisfy this requirement, Uhura must be able to update the model it used to enumerate relaxations online. Currently Uhura only accepts 'yes' or 'no', which decides if it should continuous the enumeration or terminate. If the user asks for more goals or changes the preference models, Uhura should be able to incorporate these modifications into its enumeration process decides the next alternative to present.

6.2 Summary

In this thesis we presented Uhura, a collaborative temporal plan diagnosis system. Uhura is designed to take over-constrained user goals with temporal flexibility and contingencies, specifically Qualitative State Plans, and work with the user to generate temporal relaxations that enable a complete and consistent temporal plan to be generated that achieves a relaxed set of goals. We frame the problem of relaxing goals in QSPs as a temporal relaxation problem of inconsistent OCSTNs: all goals and durations of activities that achieve the state goals are represented as conditional temporal constraints, and the inconsistent OCSTNs are resolved through the relaxation of these constraints.

Building upon prior work on conflict-directed search techniques, Uhura introduces three innovations to address the challenges in the collaborative diagnoses of over-constrained problems: quick responses, simple interaction and small perturbation. The first innovation, quick response, is supported by the Best-first Conflict-Directed Relaxation algorithm, which enumerates minimal temporal relaxations in best-first order. This is the first method that generates minimal relaxations to over-constrained temporal problems with contingencies, making the results compact and expressive for the user. BCDR extends the Conflict-directed A* algorithm and achieves nearly two orders of magnitude improvement in run-time performance relative to the Dualize & Advance algorithm in the generation of minimal temporal relaxations, making it applicable to a larger group of real-world scenarios with hundreds of constraints.

Second, BCDR simplifies the user interaction by using minimal relaxations. Minimal relaxations are compact representations of all relaxations. It can reduce the result space of relaxation problems by several orders of magnitude. BCDR only enumerates and generates minimal relaxations, which greatly reduces the amount of information exchange required during the collaborative diagnoses process.

Third, we introduce the Continuous BCDR method that generates continuous relaxations to inconsistent temporal problems. It is a generalization of the discrete relaxation method taken by all previous approaches. Prior works take an all-or-

nothing approach in which temporal constraints are suspended in the relaxations. Continuous BCDR continuously adjusts the temporal bounds of temporal constraints in OCSTNs until the consistency is restored, hence preserves the original plan elements to the maximum. Compared to discrete relaxations, continuous relaxations avoid the unnecessary lose of utility. We presented a continuous preference function over temporal constraints that can be used to enumerate continuous relaxations in best-first order.

Uhura has been incorporated within an autonomous executive that collaborates with the operators in order to find the best alternative plans in over-constrained situations. It has been demonstrated in simulation and in hardware on a Personal Transportation System concept. In addition, Uhura has also been used in a driving assistant system to resolve conflicts in driving plans. We believe that Uhura's collaborative temporal plan diagnosis capability can benefit a wide range of applications, in both the industries and daily lives.

Bibliography

[1] `http://web.media.mit.edu/~cynthiab/research/research.html`.

[2] Teresa Alsinet, Felip Many, and Jordi Planes. Improved exact solver for weighted max-sat. In *In: Proc. of the 8th SAT conference. (2005*, pages 371–377. Springer LNCS, 2005.

[3] Alessandro Armando, Claudio Castellini, and Enrico Giunchiglia. Sat-based procedures for temporal reasoning. In *Proceedings of the 5th European Conference on Planning (ECP-1999)*, pages 97–108, 1999.

[4] James Bailey and Peter Stuckey. Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In Manuel Hermenegildo and Daniel Cabeza, editors, *Practical Aspects of Declarative Languages*, volume 3350 of *Lecture Notes in Computer Science*, pages 174–186. Springer Berlin / Heidelberg, 2005.

[5] Matthew Beaumont, Abdul Sattar, Michael Maher, and John Thornton. Solving overconstrained temporal reasoning problems. In *Proceedings of the 14th Australian Joint Conference on Artificial Intelligence (AI-2001)*, pages 37–49, 2001.

[6] Colin E. Bell. Using Temporal Constraints to Restrict Search in a Planner. Technical Report PLS-237-L, Univ. of Iowa, Iowa City, December 1984.

[7] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[8] George Dantzig. *Linear Programming and Extensions.* Princeton University Press, August 1963.

[9] Randall Davis. Diagnostic reasoning based on structure and behavior. *Artificial Intelligence*, 24:347–410, 1984.

[10] Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32:97–130, 1987.

[11] Johan de Kleer and Brian C. Williams. Diagnosis with behavioral modes. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI89)*, 1989.

[12] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence*, 49(1-3):61–95, 1991.

[13] Robert Effinger and Brian Williams. Conflict-directed search through disjunctive temporal plan networks. *CSAIL Research Abstracts - 2005*, 2005.

[14] Robert T. Effinger. Optimal temporal planning at reactive time scales via dynamic backtracking branch and bond, 2006.

[15] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345–, June 1962.

[16] Eugene C. Freuder and Richard J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58(1-3):21–70, 1992.

[17] Esther Gelle. Solving mixed and conditional constraint satisfaction problems. *Constraints*, 8:2003, 2003.

[18] Esther Gelle and Mihaela Sabin. Solving methods for conditional constraint satisfaction. In *In IJCAI-2003*, pages 7–12, 2003.

[19] I hsiang Shu, Robert Effinger, and Brian Williams. Enabling fast flexible planning through incremental temporal reasoning with conflict extraction. In *Proceedings of 2005 International Conference On Automated Planning and Scheduling*, pages 252–261, 2005.

[20] I hsiang Shu, Robert Effinger, and Brian C. Williams. Enabling fast flexible planning through incremental temporal reasoning with conflict extraction. In *Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS 05)*, pages 252–261, 2005.

[21] L. G Khachian. A polynomial algorithm in linear programming. *Dokl. Akad. Nauk SSSR*, 244:1093–1096, 1979.

[22] P. Kim, B. C. Williams, and M. Abramson. Executing reactive, model-based programs through graph-based temporal planning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-01)*, pages 487–493, 2001.

[23] Phil Kim. Model-based planning for coordinated air vehicle missions, 2000.

[24] Thomas Leaute and Brian C. Williams. Coordinating agile systems through the model-based execution of temporal plans. In *ICAPS*, pages 22–28, 2005.

[25] Steven James Levine. Monitoring the execution of temporal plans for robotic systems. Master's thesis, Massachusetts Institute of Technology, 2012.

[26] George A. Miller. Wordnet: A lexical database for english. *Communications of the ACM*, 38:39–41, 1995.

[27] Michael D. Moffitt and Martha E. Pollack. Partial constraint satisfaction of disjunctive temporal problems. In *Proceedings of the 18th International Florida Artificial Intelligence Research Society Conference (FLAIRS-2005)*, 2005.

[28] Michael D. Moffitt and Martha E. Pollack. Temporal preference optimization as weighted constraint satisfaction. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI-2006)*, 2006.

[29] B. O'Sullivan, A. Papadopoulos, B. Faltings, and P. Pu. Representative explanations for over-constrained problems. In *AAAI 07*, pages 323–328, 2007.

[30] Bart Peintner, Michael D. Moffitt, and Martha E. Pollack. Solving over-constrained disjunctive temporal problems with preferences. In *Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS 2005)*, 2005.

[31] Bart Peintner and Martha E. Pollack. Low-cost addition of preferences to dtps and tcsps. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI-2004)*, pages 723–728, 2004.

[32] Francesca Rossi, Alessandro Sperduti, Kristen Brent Venable, Lina Khatib, Paul H. Morris, and Robert A. Morris. Learning and solving soft temporal constraints: An experimental study. In *CP*, pages 249–263, 2002.

[33] Kostas Stergiou and Manolis Koubarakis. Backtracking algorithms for disjunctions of temporal constraints. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, pages 248–253, 1998.

[34] Roni Stern, Meir Kalech, Alexander Feldman, and Gregory Provan. Exploring the duality in confict-directed model-based diagnosis. In *Proceedings of the 27th National Conference on Artificial Intelligence (AAAI-2012)*, 2012.

[35] Ioannis Tsamardinos, Martha E. Pollack, and John F. Horty. Merging plans with quantitative temporal constraints, temporally extended actions, and conditional branches. In *In Proc. Intl Conf. on AI Planning and Scheduling (AIPS*, pages 264–272. AAAI Press, 2000.

[36] Sebastian Varges, Fuliang Weng, and Heather Pon-Barry. Interactive question answering and constraint relaxation in spoken dialogue systems. In *Proceedings of the 7th SIGdial Workshop on Discourse and Dialogue*, 2006.

[37] Brian C. Williams and Robert J. Ragno. Conflict-directed a* and its role in model-based embedded systems. *Journal of Discrete Applied Mathematics*, 2002.

[38] Baoshi Yan, Fuliang Weng, Zhe Feng, Florin Ratiu, Madhuri Raya, Yao Meng, Sebastian Varges, Matthew Purver, Feng Lin, Annie Lien, Tobias Scheideck, Badri Raghunathan, Rohit Mishra, Brian Lathrop, Harry Bratt, and Stanley Peters. A conversational in-car dialog system. In *Proceedings of the Annual*

Conference of the North American Chapter of the Association for Computational Linguistics (NAACL-HLT), 2007.