

**Policy Compilation for Stochastic Constraint
Programs**

by

Delia Stokes Stephens

S.B., Massachusetts Institute of Technology (2022)

Submitted to the Department of Aeronautical and Astronautical
Engineering

in partial fulfillment of the requirements for the degree of

Master of Science in Aeronautical and Astronautical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2022

© Massachusetts Institute of Technology 2022. All rights reserved.

Author
Department of Aeronautical and Astronautical Engineering
May 17, 2022

Certified by.....
Brian C. Williams
Professor of Aeronautics and Astronautics
Thesis Supervisor

Accepted by
Jonathan P. How
R. C. Maclaurin Professor of Aeronautics and Astronautics
Chair, Graduate Program Committee

Policy Compilation for Stochastic Constraint Programs

by

Delia Stokes Stephens

Submitted to the Department of Aeronautical and Astronautical Engineering
on May 17, 2022, in partial fulfillment of the
requirements for the degree of
Master of Science in Aeronautical and Astronautical Engineering

Abstract

Real-world risk-bounded planning and decision-making problems are fluid, uncertain, and highly dynamic, demanding an architecture which can encode and solve a rich set of problems involving decision-making under uncertainty. While many solution architectures exist for solving deterministic CSPs, very few are able to generate decisions that are robust to uncontrolled, stochastic events, and even fewer are able to construct conditional policies that are able to adapt online to these uncertain outcomes. In this thesis, I present a variant of the Optimal Satisfiability Problem Solver (Op-Sat) that solves dynamic, chance-constrained satisfiability problems. The proposed variant solves these real-world problems efficiently and encodes policies compactly through a hybrid architecture that (a) encodes probabilistic information explicitly as logical constraints, (b) performs temporal reasoning to extract logical temporal conflicts, and (c) compiles out the constraints of a Weighted, Conditional, Stochastic CSP into a compact policy representation which may be efficiently queried. Such an architecture facilitates the design of robust, risk-aware systems by providing a user with the ability to solve a rich set of problems involving mixed logical and temporal constraints.¹

Thesis Supervisor: Brian C. Williams

Title: Professor of Aeronautics and Astronautics

¹This research was generously supported by Airbus SE.

Acknowledgments

Wow! 5 years of MIT are coming to a close. When I started this journey in 2017, I had no idea just how *hard* it would be and how much I would have to rely on my mentors, friends, and family to get me through school.

I'd like to begin by thanking my lab, the MERS group in CSAIL, and in particular Professor Brian Williams for taking a chance on me as both my undergraduate and graduate advisor. His support was critical throughout my undergraduate and graduate careers, and I am incredibly grateful for the insight, care, and kindness he demonstrated during my time at MIT. I'd also like to thank some of my incredible labmates, in no particular order: Marlyse Reeves for being my introduction to the MERS lab, mentor, and friend; Cameron Pittman for being a great lunch buddy and test subject of daily Stata café bowls; Andrew Wang for tolerating (and even encouraging) my love for the esoteric; Eric Timmons for patiently guiding me along my Common Lisp journey; Tesla Wells for listening to complain and providing me with pizza at critical junctures; Simon Fang for helping me wrap my head around my research; Yuening Zhang for being a great mentor and TA; Sungkweon Hong for asking the kindest, most insightful questions; and everybody else for making my desk a home after a long hiatus from in-person activities.

I would also like to thank Sebastien Boria and my sponsors at Airbus for their insights and ideas which made this research possible.

MIT would not have been possible without my friends and teammates. To the entire MIT Field Hockey Team, thank you! You were a bright spot during some long days, and I am proud to call all of you my friends and teammates. Thank you also to my friends in the Air Force ROTC program who taught me how to find humor in the mundane; I am truly excited to serve with all of you. To the folks of the MIT Cycling Club— Joanna, Devin, Hannah, Sarah, Berk, and so many others — thanks for giving me a new hobby and making me feel normal when I talk about bicycles for extended periods of time. I'd also like to thank my partner, Christine Padalino, without whom none of this would have been possible. Christine, you remind me of

what is actually important when I'm feeling like *everything* is important, and your superhuman tolerance of my ridiculousness has made the past 4 years a true joy.

Finally, I'd like to thank my family. To my grandparents, thank you for shaping me into the woman I am today. To Mom, Dad, and my little brother, James, thank you for being there for me over the past 23 years of my life. James, you only get 18.5 years of thanks. Play your cards right, and you could get a few more good years. Without your guidance, support, and love, I could have never imagined myself getting any degree from MIT, let alone two! Thank you for believing in me even when I didn't, picking up the phone when I called (most of the time), and truly listening to everything I had to say. I love you all very much.

Contents

1	Introduction	15
1.1	Representative Scenario	16
1.2	Approach in a Nutshell	19
1.2.1	A Simple Example	20
1.3	Thesis Structure	22
2	Problem Statement	25
2.1	The Need for an Expressive Chance-Constrained CSP Solver	26
2.1.1	Motivating Examples	27
2.1.2	Problem Requirements	30
2.2	Formal Problem Statement	32
2.3	Chapter Summary	33
3	Approach	35
3.1	Hybrid Solver Architecture	36
3.1.1	Policy Representation	37
3.2	The Weighted CSCSP Subsolver	41
3.2.1	Conditional CSP Subsolver	42
3.2.2	Constraint Compiler	42
3.3	Bayesian Inference Subsolver	43
3.3.1	Intuition: Performing Bayesian Inference	44
3.4	Temporal Subsolver	44
3.4.1	Intuition: Extracting Temporal Conflicts	44

4	Weighted CSCSP Subsolver	47
4.1	Problem Statement: Weighted CSCSPs	48
4.2	Approach: Combining Two Solvers	50
4.3	Conditional CSP Subsolver	51
4.3.1	Problem Statement: Conditional CSPs	51
4.3.2	Introducing Another Example	52
4.3.3	Approach: Compiling out Conditional Variables	53
4.4	Constraint Compiler	55
4.4.1	Problem Statement: Constraint Compiler	56
4.4.2	Simple Example: Continued	56
4.4.3	Approach: Constructing the Policy BDD	57
5	Bayesian Network Compilation	63
5.1	Problem Statement: Bayesian Inference	63
5.1.1	Background: Probabilistic Graphical Models	64
5.1.2	Formal Problem Statement	64
5.2	Approach: Performing Bayesian Compilation	66
5.2.1	A More Complex Example	68
6	Temporal Constraint Subsolver	71
6.1	Problem Statement: Temporal Subsolver	72
6.2	Approach: The Temporal Reasoning Algorithm	73
6.2.1	Background: Graph-Based Temporal Reasoning	73
6.2.2	Modified Labeled APSP	75
7	Discussion and Future Work	83
7.1	Discussion	83
7.2	Future Work	84
A	Extensions	87
A.1	Extension: Reasoning on a Policy BDD	87
A.1.1	Background: Extracting the Best Policy from Explicit Graphs	88

A.1.2	Extracting the Best Policy from the Policy BDD	90
A.1.3	Using the Policy BDD in Online Execution	91
A.2	Extension: Solving a Planning Problem	91
B	Implementation	99
B.1	Systems Used	99
B.1.1	Odo	99
B.1.2	Riker	100
B.1.3	OpSat-v3	101
B.2	Our Simple Example	101
B.2.1	Problem Statement	101
B.2.2	Extracted Temporal Conflicts	102
B.2.3	Resultant BDD	103
B.3	Converting Between Modeling Languages	103
B.3.1	Semantic Differences	104
B.3.2	Conversion into RIKER’s modeling language	105

List of Figures

1-1	A robot reasoning online about its observations.	17
1-2	A visual representation of our mixed-logic temporal CSP. Edges denote <i>temporal constraints</i> with of the form [lower bound, upper bound], denoting the <i>temporal distance</i> between events. Controllable choices are marked by a double circle, with uncontrollable choices marked by a dashed circle.	21
1-3	The final resultant BDD for our example problem.	23
2-1	A robot reacting to an unexpected obstacle on the factory floor.	26
2-2	A valid solution to the n -queens problem.	28
2-3	A visual representation of our mixed-logic temporal CSP, wherein the human and robot collaborate to prepare the human for their commute and the human then executes their desired commute.	30
3-1	The proposed architecture of DCC-OPSAT.	38
3-2	An explicit graph for a simple problem.	39
3-3	The corresponding policy tree for a simple problem.	40
3-4	The node f_1 corresponding to the decision variable assignment $(R_1 = c_1)$	41
4-1	An enhanced view of the WCSCSP Subsolver architecture.	48
4-2	The node f_1 corresponding to the decision variable assignment $(R_1 = c_1)$	59
4-3	The resultant BDD node f_7 corresponding to the example constraint $\text{active}_{H_1} \Rightarrow ((H_1 = c_1) \Rightarrow ((R_1 = c_1) \wedge (R_2 = c_2)))$	59

4-4	The resultant Policy BDD, which compactly encodes all of our constraints.	61
6-1	A simple precedence graph. Nodes representing decision variables are annotated with their associated events.	81
7-1	An alternative encoding of the constraint ($R_1 = c_1$).	84
A-1	A simple explicit graph, with a uniform prior distribution..	89
B-1	A visual representation of our mixed-logic temporal CSP. Edges denote <i>temporal constraints</i> with of the form [lower bound, upper bound], denoting the <i>temporal distance</i> between events. Controllable choices are marked by a double circle, with uncontrollable choices marked by a dashed circle.	102
B-2	The final resultant BDD for our example problem.	104
B-3	A visual depiction of the CONSTRAINT-HELPER procedure.	108

List of Tables

1.1	The number of rivets required for each subassembly and the installation time for the subassembly. As an example, it would take a human 60 minutes to rivet and install Subassembly B, as it takes between 1 and 2 minutes per assembly and 10 minutes to attach Subassembly B. . .	18
1.2	A CPT.	21
4.1	The resultant CPT added to our influence diagram	55
5.1	The conditional probability table (CPT) for our simple influence diagram.	67
5.2	The added WMC vars	67
5.3	A more complex example to demonstrate our WMC encoding.	69
A.1	The additional propositional constraints for \mathcal{V}_{CT} and \mathcal{T}'	97
B.1	State space for our example problem.	102
B.2	Temporal constraints for the simple example problem.	103
B.3	The conditional probability table (CPT) for our simple influence diagram.	103
B.4	The added WMC vars	103

Chapter 1

Introduction

A robot’s environment is never deterministic. Instead, there is a huge demand for autonomous systems that can reason about their uncertain environments, reacting fluidly to uncontrollable decisions to quickly generate a new sequence of decisions and activities with a high likelihood of mission success. The need for flexible, reactive systems that can reason over a rich set of constraints that encode an problem’s discrete decisions, scheduling requirements, and uncertainty is apparent in a variety of contexts, including aerospace manufacturing, scientific exploration, household robotics, and large-scale distribution & logistics.

In this thesis, we argue that designing a system that spends computational resources determining the set of possible responses to the uncertainty in its environment allows us to solve an expressive set of problems in a wide variety of domains. We will use human-robot collaboration as our motivating example of an uncertain environment. Central to our approach are the following notions:

1. **Risk-Bounded Uncertainty.** Agents should understand the concepts of *risk* and *uncertainty*. By providing an agent with a *risk bound* instead of simply asking it to minimize risk, we can encode a risk tolerance that allows the agent to act even when it is unsure. Otherwise, the agent might choose to simply wait until it observes all uncontrollable decisions before springing into the action once the plan has already been executed by its uncontrollable partners.

2. **Dynamic, Conditional Policies.** Rather than asking an agent to rapidly re-compute a valid solution when it encounters an unexpected observation, it is more efficient to compile a *policy* that acts as a rulebook for our agent’s online execution conditioned on uncontrollable decisions. Our agent can then quickly turn to the section of its rulebook for the set of observed decisions, select an action that does not violate the problem’s risk bound, and continue its execution fluidly.
3. **Temporal Reasoning.** Discrete decisions affect an agent’s schedule of events; for example, deciding to refuel before traversing an uncertain environment might make the traversal take longer due to the increased weight, but the traversal is less likely to fail due to a fuel shortage. Our hybrid solver should be able to reason about which sets of choices yield *temporal infeasibility* so that it may avoid those execution paths online.

The constraint satisfaction and temporal planning communities have developed a rich set of tools that can reason on a problem’s logical satisfiability [18], its temporal causality [4], and controllability [17]; however, there are few that *combine* the notion of satisfiability and temporal planning to provide both communities with a tool for generating expressive, dynamic policies in uncertain environments. This thesis leverages the key insights from both communities, developing a hybrid solver capable of generating conditional, conditional policies that encode an agent’s flexible responses to online observations. By providing both communities with a unified modeling language and solver, we enable a broader set of problems to be solved quickly and accurately with conditional policies that allow an agent to efficiently reason online.

1.1 Representative Scenario

To motivate our hybrid solver, we consider an aerospace manufacturing scenario (modified from [11]) in which a robot needs to react to its uncertain environment on-the-fly. In this scenario, a human is collaborating with an assembly robot to build an air-

craft wing within 40 minutes. Broadly speaking, the wing panels must be secured while the agents rivet the panel into place. After the panels are secured, one of three subassemblies may be bolted onto the wing section. Each subassembly requires a different adhesive—Subassembly A requires bolts, Subassembly B requires aerospace-grade adhesive, and Subassembly C is simply wing-walk tape pasted onto the wing surface itself. Only one subassembly must be affixed to the wing. The human can choose to perform the riveting themselves, in which case the robot should fetch the required rivets and hand them to the human.

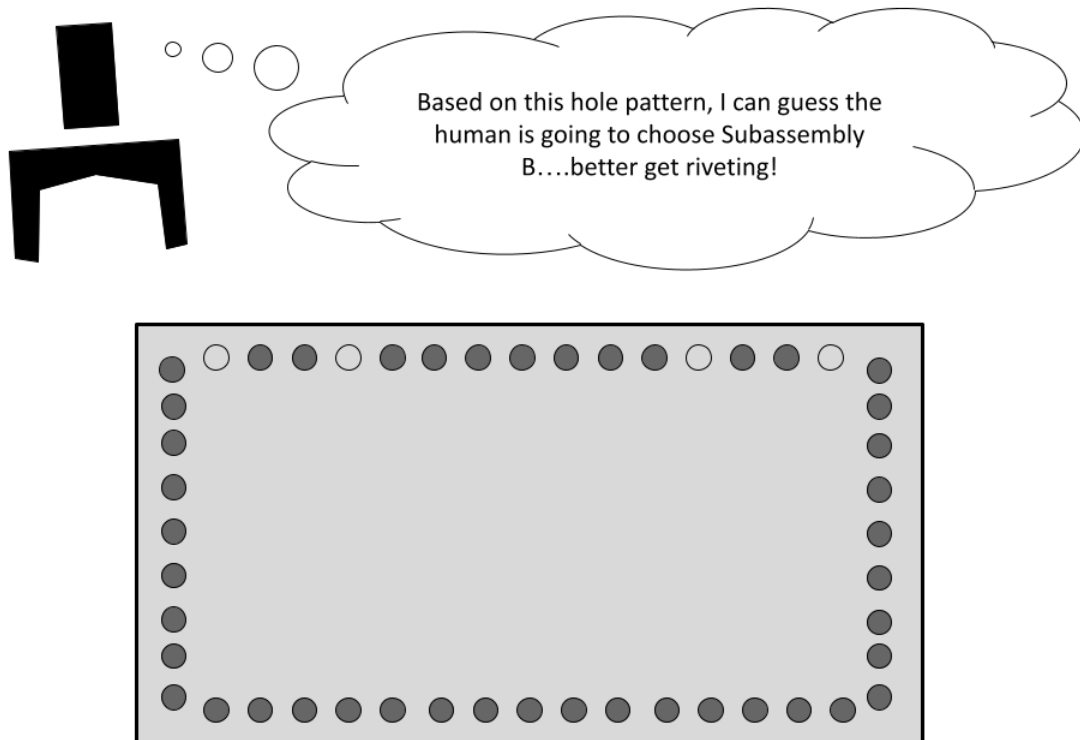


Figure 1-1: A robot reasoning online about its observations.

To further complicate things, the human-robot duo has 40 minutes to perform the assembly task, and each of the rivets takes between one and two minutes to perform. The number of rivets and their pattern depends on which subassembly will be bolted onto the wing. That is to say, the robot can observe the rivet pattern, infer the total number of rivets required, and start riveting itself if it thinks that the human will

not finish the task in time. Otherwise, the robot should simply observe the human’s riveting and provide them with the rivets for their rivet gun. As you can see from the durations listed in Table 1.1, we would expect to see the robot intervene if it believes Subassembly B is the active subassembly, grabbing a rivet gun for itself and collaborating with the human, as it takes the human 50 minutes to install the rivets and 10 minutes to install the subassembly. In this case, the human and robot should not rivet at the same location. Similarly, if the robot infers that Subassembly C is the active subassembly, it should intervene to help out the human before temporal constraints are violated.

Subassembly	Number of Rivets	Subassembly Installation Time
A	10	20
B	50	10
C	30	15

Table 1.1: The number of rivets required for each subassembly and the installation time for the subassembly. As an example, it would take a human 60 minutes to rivet and install Subassembly B, as it takes between 1 and 2 minutes per assembly and 10 minutes to attach Subassembly B.

Finally, we bound the robot’s risk tolerance. Suppose the robot has 90% confidence the human will choose Subassembly A, 8% it will choose Subassembly B, and 2% Subassembly C. If we set the robot’s risk tolerance conservatively (for example, 0.1%), then the robot will not pick up bolts until it is quite certain that the robot will choose Subassembly A. The robot’s confidence can change over time as it observes the human’s riveting pattern. If the robot’s confidence grows such that it is 99.9% confident the human will choose Subassembly A, then it can fetch the bolts to prepare the human for the next task! However, picking too early is *risky*; if it fetches the incorrect Subassembly or fails to help the human when they need it, the assembly will not be completed within the allotted 40 minutes.

Note just how *expressive* our representative scenario is. It mixes **temporal** constraints (the human-robot due has 40 minutes to complete the assembly), **logical** constraints (the human and robot may not rivet the same location), and a **risk** constraint (the robot may not act until its confidence exceeds a predetermined threshold).

This scenario also encodes the desirable quality of flexibility. If the robot observes the human grab the rivet gun, then it should be able to quickly infer the human’s intent and fetch the rivets, rather than going through an extensive re-planning process with the updated information. This desire for fast online reasoning promotes the concept of a **policy**. The robot should be able to use an efficiently-represented operations manual to determine how it should act in uncertain scenarios.

1.2 Approach in a Nutshell

By coordinating techniques from the constraint programming and planning communities, we propose a dynamic, chance-constrained hybrid solver that allows us to model an expressive set of problems and generate dynamic conditional policies. By breaking down a problem into smaller subproblems, we are able to leverage the state-of-the-art from both communities, yielding a conditional policy that correctly encodes an agent’s set of responses to its uncontrollable environment. Our hybrid solver, dynamic, chance-constrained OpSat (dcc-OpSat), extends the Optimal Satisfiability solver OpSat [18] and its chance-constrained variation cc-OpSat [6] to generate a *dynamic, chance-constrained policy* that allows an agent to reason flexibly on uncontrollable decisions.

We achieve this task by dividing the problem into three central subproblems:

1. **Policy Compilation.** dcc-OpSat’s desired goal is to supply a conditional policy over the discrete-valued observations extracted from the problem’s set of constraints. This policy is conditioned on a history of observations (assignments to decision variables by the agent and its environment) and efficiently represents a set of possible assignments for fast online reasoning. Typically, a policy is represented as a tree-like structure. Here, we leverage the insight that we may build this tree up from the constraints themselves, rather than finding a whole family of solutions and compressing out duplicated structures. This ground-up approach is more space-and-time efficient than storing the entire explicit graph or reducing the explicit graph into a more efficient representation.

2. **Probabilistic Reasoning.** Our representative example demonstrated the need for an agent to perform fast probabilistic reasoning about its environment. For example, if the manufacturing agent takes too long to perform its probabilistic inference about the likelihood that the human is attaching Subassembly B, its slow computation process might cause a delayed intervention.

3. **Temporal Reasoning.** Our hybrid solver is capable of performing temporal reasoning on a set of temporal constraints, extracting the set of decisions that will never be feasible. We can encode infeasibility into our policy of feasible solutions by representing that infeasibility as an additional constraint. As an example, it will never be temporally feasible for the human to do the riveting for Subassembly B by themselves; instead, the robot *must* choose to assist them. We could encode this temporal conflict logically with $\neg(\text{assembly-b} \wedge \neg\text{robot-help})$.

We represent this policy as a compact, efficient form of a decision tree. A decision tree branches on the possible choices an agent can make or observe; following the decision tree from root to leaf gives us a partial assignment to decision variables. We can leverage the policy’s efficient structure for performing probabilistic inference on our problem during online computation. By weighting the nodes in our tree with their relative likelihoods, we can find the *least risky* execution that satisfies our risk bound.

1.2.1 A Simple Example

In this section, we’ll introduce a simple example to guide the rest of this thesis, introducing at a high-level the notion of a temporal network, logical constraints, and probabilistic reasoning.

Example 1. Consider a human and robot working together to accomplish two tasks in sequence. First, the robot chooses to perform one of two tasks (c_1 or c_2). Task c_1 takes between 10 and 20 minutes, but task c_2 only takes between two and four.

Then, the human chooses to perform one of two tasks (c_1 or c_2), each of which *also* takes between two and four minutes. If the human chooses to perform task c_1 , then the robot must as well. The human-robot duo must complete their tasks in less than 10 minutes.

We may model our problem with the following state space:

$$\mathcal{V} = (\mathcal{V}_C = \{R_1\}) \cup (\mathcal{V}_U = \{H_1\})$$

where both H_1 and R_1 have domains $D_{H_1, R_1} = \{c_1, c_2\}$. Note that H_1 is uncontrollable. As a result, we want to define the *prior likelihood* of variable H_1 , which we do with the following influence diagram:

$H_1 = c_1$	0.1
$H_2 = c_2$	0.9

Table 1.2: A CPT.

The temporal constraints, though omitted for brevity, can be represented graphically:

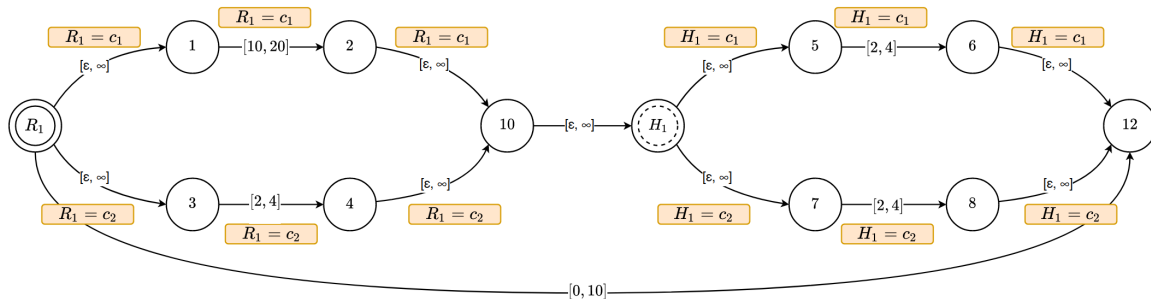


Figure 1-2: A visual representation of our mixed-logic temporal CSP. Edges denote *temporal constraints* with of the form $[\text{lower bound}, \text{upper bound}]$, denoting the *temporal distance* between events. Controllable choices are marked by a double circle, with uncontrollable choices marked by a dashed circle.

In Figure 1-2, each temporal constraint is *guarded* by the assignment to the decision variable. Guards are shown highlighted in orange. Events, which denote remarkable points in time, are shown by nodes and edges denote temporal constraints. As an example, the temporal constraint that encodes the action corresponding with

robot task c_1 may be defined as follows:

$$\text{STC}(e_1, e_2, 2, 4, \psi_c = (R_1 = c_1))$$

which takes on the following more formal meaning:

$$(R_1 = c_1) \Rightarrow (l \leq e_2 - e_1 \leq u)$$

Similar temporal constraints are included for every edge that appears in our candidate graph, though they are not included in the description for brevity. In addition to the temporal constraints, we must model that the human’s choice influences the robot’s, which may be accomplished by encoding the following constraint into our simple example:

$$\mathcal{C} = \{(H_1 = c_1) \Rightarrow (R_1 = c_1)\}$$

Intuitively, we can tell that our human-robot pair has a 90% chance of succeeding at completing both tasks within the allotted time. This is because if the human chooses to complete task c_1 , then the robot must also complete task c_1 (which takes more than the allotted 10 minutes), violating the problem’s temporal constraints. Our *conditional policy* encodes the notion of a history of assignments to decision variables and can be represented in a tree-like structure called a Reduced, Ordered Binary Decision Diagram as seen in Figure 1-3.

In the next chapters, we’ll build up to this intuitive result and represent the robot’s conditional choices as a Policy BDD. Finally, in Appendix B.2, we will explicitly go through every step of the process, discussing how to model and implement this problem using the tools described this thesis.

1.3 Thesis Structure

In Chapter 2 of this thesis, we outline the defining characteristics of our problem statement, highlighting a few representative scenarios that motivate the design of our

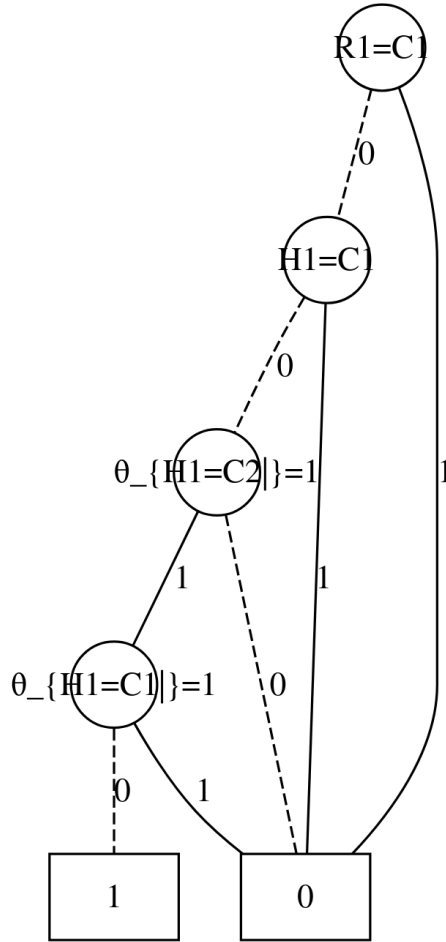


Figure 1-3: The final resultant BDD for our example problem.

hybrid solver. Then, in Chapter 3, we break down our problem statement into the relevant compilation, Bayesian inference, and temporal reasoning subproblems, discussing how each subproblem is coordinated to yield a policy that fully represents an agent’s possibilities. In Chapter 4, we discuss the policy compilation problem, defining key characteristics of our inputs and outputs and outlining a detailed approach that goes from a weighted, conditional stochastic logical problem to a policy BDD that efficiently encodes an agent’s responses. Then, in Chapter 5, we present the Bayesian inference subsolver, discussing its formal problem statement and relevant insights. During this chapter, we introduce the human-robot collaboration extension from [11], discussing how to extract the “best” possible set of decisions in response to a partial assignment of variables subject to a chance constraint. This extension

grounds our Bayesian inference in a real-world application, though the formulation proposed by dcc-OpSat is more flexible and allows for the implementation of a wide range of algorithms over the Policy BDD. Chapter 6 discusses how we mix in temporal constraints with our logical constraints, highlighting key algorithms from [11] and [4] that allow us to extract a set of logical conflicts from a network of temporal constraints. The conflicts extracted from the temporal constraints are then passed back to our parent solver and folded into the Weighted CSCSP for compilation into a policy BDD.

Chapter 2

Problem Statement

In this chapter, we will build up the intuition for our problem statement, identifying key elements of interesting problems that our hybrid solver, DCC-OPSAT, can accept. In short, we identify the growing need for a hybrid solver that combines the best elements of the SAT and temporal planning communities, enabling us to express a rich set of constraints useful for a broad range of multi-agent problems in uncertain environments. Then, we describe the key characteristics of the problems our hybrid solver can manipulate. Finally, we turn those characteristics into a formal problem statement for an architecture that can solve a rich set of problems for risk-bounded, multi-agent planning and decision-making problems.

An agent's environment is rarely perfectly deterministic. Instead, an autonomous agent must be able to react flexibly to the uncontrollable choices made by the agents around it and its environment itself. However, solving multi-agent, risk-bounded problems on-the-fly is computationally intensive; instead, it is often advantageous to provide an agent with a rulebook for its behavior, or *policy*, which allows it to quickly make decisions once it observes uncontrollable choices.

Consider the intuitive example of a robot traversing a factory floor, going from waypoint to waypoint to retrieve tools and provide them to a human partner. If the robot encounters an unexpected obstacle in its path, it should not spend excessive time reformulating a planning problem, considering all possible schedules that accomplish its goals, and picking the best one. Instead, the agent should be able to quickly consult

a policy generated during its pre-compilation procedure, determine which response best meets its criteria for success, and follow the path around its obstacle that gets it to the next tool in the factory.

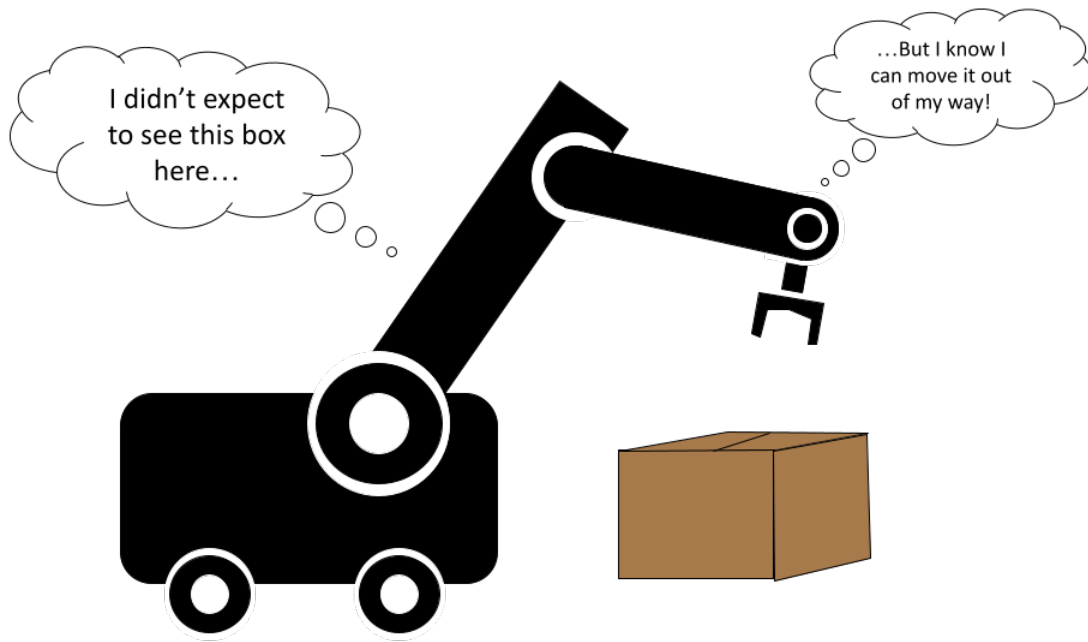


Figure 2-1: A robot reacting to an unexpected obstacle on the factory floor.

2.1 The Need for an Expressive Chance-Constrained CSP Solver

Many solution architectures exist for solving deterministic CSPs [18], returning a single assignment to the decision variables that satisfies a given set of constraints. However, not all problems are deterministic; in situations where an agent is acting in an uncontrollable environment, it becomes necessary to solve for a policy that encodes an agent's flexible response to uncontrollable decisions. Additional work has

been done to create solvers for the CSP’s stochastic [17] and conditional [7] variants, returning a more flexible policy that represents a set of possible assignments given uncontrollable observations. Similarly, there are solution architectures for extracting temporal conflicts from a set of temporal constraints [4]. However, no hybrid solver exists that combines the notions of temporal and conditional constraints to yield a dynamic, conditional policy, allowing a user to encode a rich set of multi-agent decision-making problems.

In this thesis, we leverage key insights from the constraint programming and temporal planning communities, developing a dynamic, chance-constrained Optimal Satisfaction Solver (DCC-OPSAT) that is capable of manipulating a set of logical, temporal, and state-based constraints to return a *dynamic* policy governing an agent’s behavior in an uncontrollable environment. By coordinating between existing sub-solvers that operate on a subset of the constraints and variables, we are able to pull from the state of the art in a variety of communities, encoding a rich set of problems useful for multi-agent domains.

This hybrid architecture combines three critical aspects of real-world problems. First, it can encode **stochastic and controllable** discrete-valued variables representing choices made by an agent or its environment. Additionally, variables may be **conditioned** on an agent’s observations. Finally, it combines **temporal and logical** constraints. When all three characteristics are combined, we have a hybrid solver that generates dynamic policies for risk-bounded behavior in uncertain environments.

2.1.1 Motivating Examples

We now consider a few examples that highlight key characteristics of our problem statement.

Stochastic n -Queens

Here, we discuss an example that illustrates critical features of a chance-constrained CSP *without* temporal constraints. Simply put, neither our agent nor its environment

is time-constrained, so there is no need to reason on temporal constraints.

Example 2. Consider a modified version of the canonical n -queens problem. In the traditional expression, an agent is tasked with placing n queens on an $n \times n$ chessboard such that no two queens attack each other. In our modified formulation, an agent and a human are *collaborating* to place n queens on the chessboard; the autonomous agent has no control over the human’s choices but should take actions in an “assistant” role. The human moves first; after the human’s move, the robot and human take turns placing pieces on the chessboard until it is filled. The robot knows a little about how the human is expected to place pieces; for example, if the human places its first piece in position $(2, 0)$, it is much less likely to place a piece there in the future. Central to this formulation is the notion of *stochasticity*: our autonomous agent lacks total control over its environment and must instead respond to actions outside of its control. Additionally, our formulation includes the notion of a *chance constraint*; if a robot’s choice is expected to decrease the probability of success below a threshold, then it may not make that choice. If no possible set of assignments in the policy exists satisfying the chance constraint, then it should signal an error to alert its human partner.





(0, 0)	(0, 1) 	(0, 2)	(0, 3)
(1, 0)	(1, 1)	(1, 2)	(1, 3) 
(2, 0) 	(2, 1)	(2, 2)	(2, 3)
(3, 0)	(3, 1)	(3, 2) 	(3, 3)

Figure 2-2: A valid solution to the n -queens problem.

The chance-constrained formulation is able to encode the notion of *risk*. For

example, if the robot expects that the human’s next move will be placing a piece in position $(3, 2)$, then it should not place a piece in positions $(3, 1)$ or $(2, 1)$ in order to increase the likelihood of mission success. Finally, our formulation also contains an inherent *assignment ordering*; that is, the human makes the first assignment, followed by the robot, with the pattern proceeding as the board is filled.

Because not all assignments are controllable by our agent, a single solution satisfying the constraints does not help our agent make informed choices about its environment. Instead, the agent should reason flexibly on the intent of its human partner. The *flexibility* demands a solution in the form of a *conditional policy*. This policy is conditioned on assignments to stochastic variables and imposes assignments to the controllable variables while respecting the input assignment ordering.

Human-Robot Collaboration

We borrow an example from [11] to demonstrate our hybrid solver’s ability to solve critical problems in human-robot collaboration.

Example 3. Consider a robot and human collaborating to prepare the human for a busy day at work, represented graphically in Figure 2-3. First, the human/robot duo must eat breakfast and fetch the required items to travel to work. Next, the human travels to work. The human can choose to sleep in or wake up early and then later chooses to eat a fueling breakfast of eggs or a lighter cereal. The robot is not sure of the human’s decisions beforehand, but it does know that the human tends to run or bike to work if they eat eggs and sleep in when they drive. The full relationship between these variables is encoded by a set of conditional probability tables that, in turn, may be expressed as an *influence diagram*, a generalization of a Bayesian network. In this way, the robot’s observations may give it more information about what a human is likely to do later in the plan. While the human wakes up and eats, the robot can choose to do at most three of the following: pump the human’s bike’s tires, grab the human’s car keys, grab the human’s shoes, or grab the human’s umbrella. Due to the problem’s *temporal constraints*, the robot may not fetch all four items.

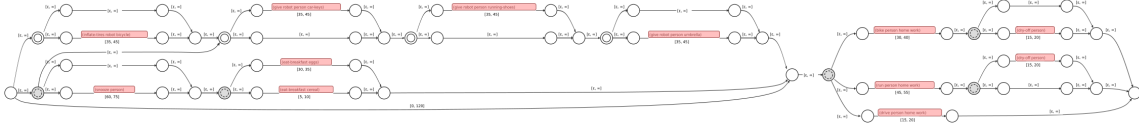


Figure 2-3: A visual representation of our mixed-logic temporal CSP, wherein the human and robot collaborate to prepare the human for their commute and the human then executes their desired commute.

After the human awakens and eats breakfast, they choose to bike, run, or drive to work. If they bike or run and it’s raining, they must dry off before sitting in their plush desk chair. Of course, a human may not bike to work if their bike has flat tires; they may not run without running shoes; and they may not drive without their car keys, so the robot has to select the “correct” pre-computed tasks to complete given its knowledge about the weather, the human’s general tendencies, and their choices thus far. As an example, the robot may use the observation that the human has chosen to sleep in to infer that they are less likely to bike or run to work, prioritizing grabbing the human’s keys over inflating the bicycle’s tires.

The morning routine problem demonstrates the need for an encoding that can represent *temporal constraints* that relate the scheduling of *events* alongside *logical constraints*. As an example, the robot’s only has time to do three of its four morning tasks, which affects the way the human behaves down the road. Furthermore, it re-emphasizes the importance of *conditionality*. For example, the robot is only forced to make certain decisions once it has observed particular human choices.

2.1.2 Problem Requirements

Our examples highlight several demands of our problem’s input and output; though there exist solvers ([7], [18], [17]) that satisfy a *subset* of these requirements, none provide a flexible architecture for dynamically solving chance-constrained CSPs.

Inputs

1. **Finite Choices.** Our problem’s *stochastic and decision variables* have discrete, finite domains. For example, in the stochastic *n*-queens problem, the human

may choose to place their first queen in Squares $(0, 0)$, $(1, 0)$, $(2, 0)$, or $(3, 0)$.

2. **Real-Valued Temporal Events.** When encoding a planning problem with temporal constraints, we must reason over variables with real-valued domains called *events* that denote critical points in times. Here, we assume that all events are *controllable* and therefore *decision* variables.
3. **Ordering.** Assignments to variables are *ordered*. For example, the human chooses where to place their chess piece first.
4. **Conditionality.** Certain assignments *activate and deactivate* other constraints. For example, if the human chooses to bike to work, they activate the set of temporal constraints that encode the biking and potential dry-off procedure.
5. **Expressive Constraints.** A problem's constraints may be:
 - **Logical:** if the human places a chess piece in Square $(0, 0)$, then the robot may not place a piece in Square $(0, 1)$.
 - **State-based:** the robot must actually be pumping the human's bike between the events corresponding to pumping.
 - **Temporal:** the human must get to work on time.
 - **Probabilistic:** the human is not likely to place a piece in Square $(0, 0)$.
 - **Risk-based:** if the probability of success falls below a certain threshold, then the solution is not correct.

Outputs

1. **Conditionality.** Our output must be *conditioned* on assignments to uncontrollable variables in order to encode the flexibility required from an uncontrollable problem.
2. **Compact.** The solution must efficiently encode the set of all possible assignments to make reasoning on assignments to uncontrollable decision variables computationally tractable.

2.2 Formal Problem Statement

In this section, we define our formal problem statement for a hybrid solver meeting the requirements outlined in Section 2.1.2.

Inputs

- \mathcal{V} , a set of variables. Each variable $v_i \in \mathcal{V}$ is associated with a *domain* D_{v_i} , which may be finite or real-valued. We further divide \mathcal{V} into the following categories to fully express our desired set of problems:
 - $\mathcal{V}_D \subseteq \mathcal{V}$, the set of discrete, finite-domain variables. A variable $v \in \mathcal{V}_D$ may be either *controllable* or *uncontrollable* (in which case $v \in \mathcal{V}_U$), and it may be either *conditional* (in which case $v \in \mathcal{V}_C$) or *unconditional*. A variable may be both uncontrollable and conditional.
 - $\mathcal{V}_C \subseteq \mathcal{V}$, the set of conditional variables. Conditional variables may be *active* or *inactive*, and they may have real or finite domains. If they are inactive, then they need not take a value from their domain and all constraints involving the inactive conditional variables are vacuously satisfied.
 - $\mathcal{V}_U \subseteq \mathcal{V}_D$, the set of uncontrollable variables.
 - $\mathcal{V}_R \not\subseteq \mathcal{V}_D \subseteq \mathcal{V}$, the set of variables with real-valued domains. These variables correspond to *events* in a temporal constraint network. Variables with real-valued domains may not be uncontrollable, but they may be conditional.

Our hybrid solver must be able to handle problems that include *at least one* of the above variable types; valid problems need not include all variable types.

- \mathcal{C} , the set of constraints. Constraints may be either:
 - *State Logical* (\mathcal{C}_L), in which case they are propositional formulae whose propositions are assignments to discrete variables.

- *Temporal* (\mathcal{C}_T), in which case they express upper and lower bounds for the relative values of the real-valued variables \mathcal{V}_R .

Constraints may also be *both* logical and temporal. Grounding this notion in Example 3, the human choosing to run to work *activates* the temporal constraint that driving takes between 15 and 20 minutes. This may be represented by the logical implication, e.g. `(human drives) \Rightarrow STC($e_1, e_2, 15, 20$)`.

- $\mathcal{C}_{\text{active}}$, a set of activation constraints expressed as guards on finite domain variables and simple temporal constraints.
- $\mathcal{V}_{\text{order}}$, an ordering over the finite-domain variables.
- Δ , which encodes the risk bound of our problem’s likelihood of success.
- Pr, which encodes probabilistic constraints on assignments to finite-domain decision variables in the form of a Bayesian network or its generalization, the influence diagram.

Outputs

- A compact encoding of the *decision tree* that forms the set of all possible assignments to decision variables. In this thesis, we choose a *Reduced, Ordered Binary Decision Diagram* for its canonicity and efficiency.

2.3 Chapter Summary

In this chapter, we extended our motivating examples to flesh out two interesting, representative scenarios—the stochastic n -queens problem and the human-robot collaboration problem. From these examples, we identified a set of critical problem features that can be solved individually but are challenging to reason over when combined. From the informal descriptions of these features, we distilled a formal problem statement that drives the architecture of our hybrid solver.

Chapter 3

Approach

With our problem statement in mind, we may begin to design a hybrid solver architecture that is capable of solving a mixed-logic temporal problem with probabilistic constraints. Because each component of our problem statement is relatively well-understood ([17], [4], [11], [7]) but the sum total—a stochastic, chance-constrained CSP with mixed logical and temporal constraints—remains an area of active research, we leverage the notion of a *hybrid* solver. A hybrid solver coordinates between a set of subsolvers, each of which can solve one of the subproblems and return useful information to its parent solver. In the DCC-OPSAT case proposed in this thesis, we identify four critical subsolvers which solve their corresponding subproblems to create efficient, compact policy representations:

1. A **Weighted CSCSP Subsolver**, which takes in a Weighted Conditional, Stochastic CSP and compiles out the constraints into a representation that may be efficiently queried. In turn, the Weighted CSCSP subsolver depends on:
 - (a) A **Constraint Compiler**, which turns the weighted, stochastic CSP and compiles out the constraints into a conditional policy.
 - (b) A **Conditional CSP Subsolver**, which compiles out the conditional variables and constraints from a Weighted CSCSP and returns a Weighted SCSP.

2. A **Bayesian Inference Subsolver**, which explicitly represents the probabilistic information from an SCSP as a set of auxiliary constraints.
3. A **Temporal Constraint Subsolver**, which gathers a set of guarded temporal constraints and yields a set of logical constraints.

Intuitively, subsolvers (2) and (3) create an augmented CSP which may be compiled into a Policy BDD by subsolver (1). Each subsolver leverages existing techniques and algorithms, which are then coordinated through the parent solver, yielding a dynamic conditional policy which represents all possible responses to a set of uncontrollable choices.

3.1 Hybrid Solver Architecture

Our hybrid solver is responsible for coordinating its subsolvers to ensure correctness. Here, we describe the high-level processes for coordination between our three subsolvers.

First, we take an input of the form described in our formal problem statement from Section 2.2. Our parent solver then does the following:

1. Hands off the subset of temporal constraints to the temporal subsolver, yielding a set of temporal conflicts.
2. If the problem contains no conditional variables and constraints, the parent solver immediately hands off the probabilistic information to the Bayesian inference subsolver, yielding an auxiliary set of variables and constraints.
3. Takes in the auxiliary variables, probabilistic constraints, and temporal conflicts to the constraint compiler, which in turn:
 - (a) “Compiles out” conditional constraints, yielding a problem that explicitly represents variable’s activation status. This modifies the probabilistic information. The modified conditional probability tables which contain our

explicit activation constraints is then passed to the Bayesian inference subsolver, which generates an additional set of variables and constraints which are then passed back into the Weighted SCSP problem.

- (b) Compiles the unconditional, weighted, stochastic CSP into a Reduced, Ordered, Binary Decision Diagram (ROBDD) representing a policy.

In short, our subsolvers generate additional variables, constraints, and weights which are added to our original problem statement; from there, we take the resultant Weighted CSCSP and yield a Policy BDD. Figure 3-1 shows the overall architecture for our hybrid solver. In orange are the problem inputs; the inputs are distributed to the appropriate subsolver, which then generates a set of auxiliary constraints and returns them to the Weighted CSCSP subsolver. The Weighted CSCSP subsolver, in turn, compiles out conditional variables to create an unconditional problem and then represents the set of solutions to that problem as a Reduced, Ordered Binary Decision Diagram with a set of weights on its nodes.

Our hybrid solver may be described formally by the following high-level algorithm which coordinates the subsolvers:

Algorithm 1 DCC-OPSAT

Input: $\langle \mathcal{V}, \mathcal{C}, \mathcal{V}_{\text{order}}, \Delta, \text{Pr} \rangle$ as defined in Section 2.2.

Output: An ROBDD and set of weights \mathcal{W} .

- 1: **if** *the problem contains temporal constraints* **then**
 - 2: Extract the temporal conflicts as \mathcal{C}_{TC}
 - 3: **end if**
 - 4: **if** *the problem contains conditional constraints* **then**
 - 5: Compile out the conditional constraints
 - 6: Modify Pr with the auxiliary conditional variables
 - 7: **end if**
 - 8: $\mathcal{C}_B, \mathcal{V}_B, \mathcal{W}_B \leftarrow \text{BAYESIAN-INFERENCE}(\text{Pr}, \Delta)$
 - 9: $\text{ROBDD}, \mathcal{W} \leftarrow \text{COMPILE-CONSTRAINTS}((\mathcal{V} \cup \mathcal{V}_B), (\mathcal{C} \cup \mathcal{C}_B \cup \mathcal{C}_{TC}), \mathcal{W}_B)$
-

3.1.1 Policy Representation

To understand our desired output, we introduce the notion of an *explicit graph* or *decision tree* to build up to our definition of a Reduced, Ordered, Binary Decision

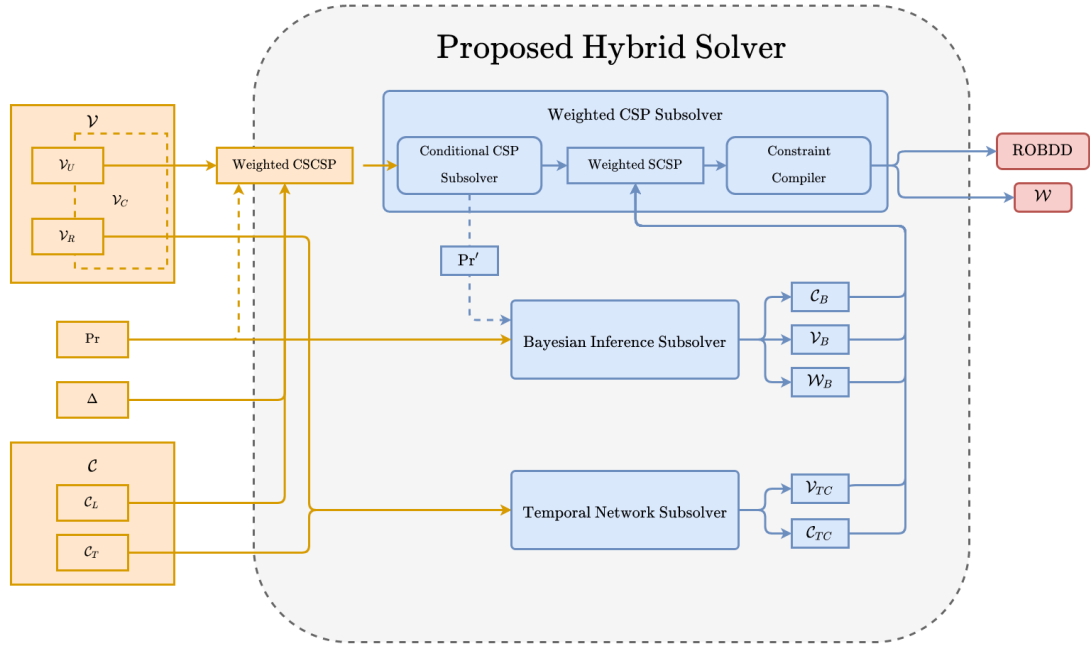


Figure 3-1: The proposed architecture of DCC-OPSAT.

Diagram.

Explicit Graph

An explicit graph represents the set of all possible policies by encoding all possible assignments to the state space of variables. Nodes in the explicit graph are the variables in \mathcal{V} and edges are assignments to those variables. Each leaf node is labeled 0 or 1, and each path from root to leaf represents an interpretation ψ_S ; if this interpretation ψ_S is a *solution* (or *model*), it satisfies all the constraints in \mathcal{C} . We define the indicator function $\mathbf{1}_{\text{sat}}$ which takes the value of 1 if ψ_S is a solution to our problem and 0 otherwise.

Explicit graphs are often represented in an AND-OR tree structure where the variables in \mathcal{V}_U represent the AND nodes, and \mathcal{V}_C form the OR nodes. In a policy tree, we choose one of assignment for each of the OR nodes. Intuitively, this is because our agent cannot select the assignment of an uncontrollable variable and must choose the “best” possible response to an uncontrollable choice. Note that, as before, our variables have discrete, finite domains. We define the set of all interpretations of a

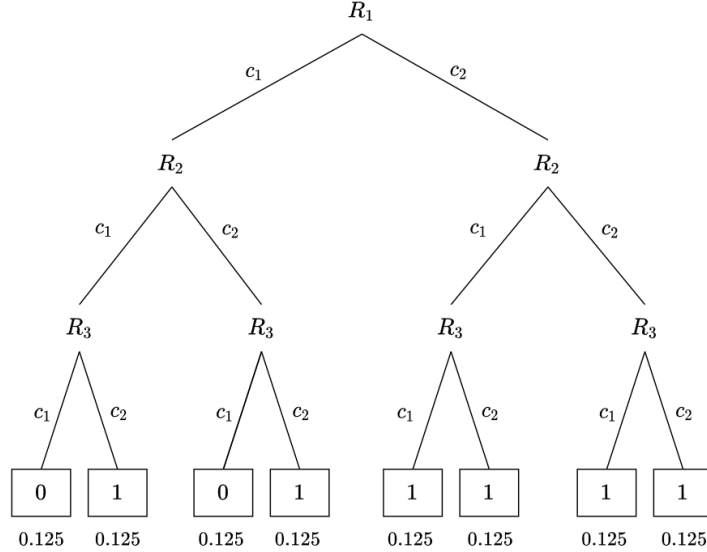


Figure 3-2: An explicit graph for a simple problem.

policy tree (the full assignments to the variables in \mathcal{V}) to be $\mathcal{S}(\pi)$.

To go from the explicit graph to our AND-OR policy tree, we traverse the explicit graph and choosing one value for each OR node. In Figure 3-3, the red nodes correspond to the uncontrollable variables. A policy is successful if its assignments satisfy the problem’s constraints; if there is stochasticity in the problem and a risk bound is provided, then the assignment must also satisfy the problem’s risk bound.

Policy BDD

A Binary Decision Diagram (and, in this case, a Reduced, Ordered Binary Decision Diagram) is a compact way of encoding Boolean functions that can be easily generated, modified, and reasoned on by graph algorithms. We choose to represent our problem output as a Reduced, Ordered Binary Decision Diagram. In an ROBDD, Boolean functions are represented by directed, acyclic graphs. Note that a policy represented by an AND-OR tree (or explicit graph) is also a logical formula over a set of observations. That is, the explicit graph in Figure 3-3 can be considered a set of boolean functions. However, the explicit graph has substantial duplicated structure as subgraphs are often repeated during the a traversal, demanding a more efficient formalization like the ROBDD which eliminates duplicated structures to yield a more

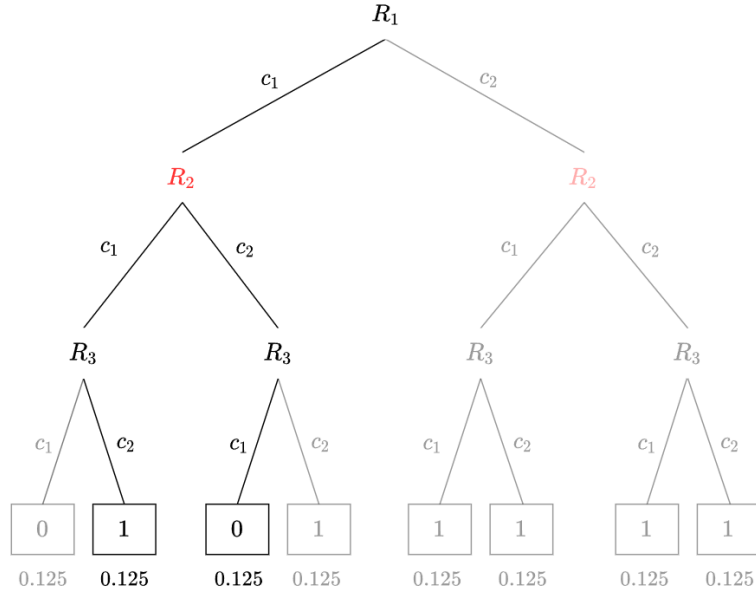


Figure 3-3: The corresponding policy tree for a simple problem.

compact representation.

Definition 1. (Reduced, Ordered Binary Decision Diagrams (ROBDD)). A *reduced, ordered BDD (ROBDD)* is a rooted, directed, acyclic graph that represents a Boolean logic formula f . The graph \mathcal{G} consists of a root node, decision nodes (which represent variable assignments), and two leaf nodes, which denote whether or not formula f is satisfied given a path from root to leaf.

Each node (except for the two terminal nodes) has two children; the edge corresponding to the 1 or “high” child of node n corresponds to the assignment in node n holding (and vice versa). A full path from root to leaf represents a full assignment to variables in the formula f . If the leaf (or *terminal*) node takes the value of 1, then the formula f evaluates to \top given the assignments along the path from root to leaf; if the terminal node takes the value of 0 the f evaluates to \perp given the assignments along the path.

We may represent an assignment to a decision variable as a single BDD node; its “high” (1) child represents the assignment holding, and the “low” (0) child represents the assignment *not* holding. Consider the variable R_1 with domain $\{c_1, c_2\}$. We represent the assignment of R_1 to value c_1 with the BDD in Figure 3-4. Note that a

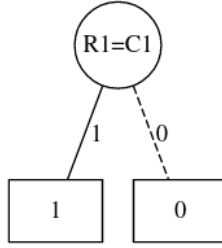


Figure 3-4: The node f_1 corresponding to the decision variable assignment $(R_1 = c_1)$.

BDD is inherently a recursive data structure; that is, a BDD node is represented by the node itself and its high and low children nodes.

Definition 2. (Policy BDD). A *Policy BDD* is simply a Binary Decision Diagram representing a policy.

3.2 The Weighted CSCSP Subsolver

In Chapter 2, we established the need for a subsolver which can handle a conditional, stochastic formulation with a set of weights on the variables, yielding a dynamic policy which compactly encodes the set of possible solutions to our problem. To formally describe our problem’s input, we define the Weighted, Stochastic Constraint Satisfaction Problem, which is composed of a set of controllable, uncontrollable, and conditional variables and constraints over those variables. The Weighted CSCSP subsolver is responsible for compiling out the set of constraints represented by a structure we call a *Weighted CSCSP* into a policy. This policy may be efficiently queried to determine which solution maximizes an objective function while satisfying the problem’s constraints.

Simply put, our Weighted CSCSP subsolver goes from a Conditional, Stochastic CSP with weighted assignments to decision variables and encodes the set of constraints \mathcal{C} as a policy. Note that our chosen policy representation is a Reduced, *Ordered* Binary Decision Diagram. Therefore, our input must also encode a variable ordering which specifies the order in which assignments are made to variables.

Intuition: Weighted CSCSP Subsolver

The Weighted CSCSP subsolver accomplishes its compilation task by in turn leveraging two subsolvers: one (Section 3.2.1) which takes in a Weighted Conditional SCSP and compiles out the conditional constraints, yielding a Weighted SCSP; and another (Section 3.2.2) which takes in the Weighted SCSP and compiles out the unconditional constraints into a Reduced, Ordered Binary Decision Diagram with a set of weights.

3.2.1 Conditional CSP Subsolver

A Reduced, Ordered Binary Decision Diagram cannot reason on *conditional* variables and constraints. For that reason, our hybrid architecture must leverage a subsolver proposed in [11] to *compile out* the conditional nature of a problem into an unconditional CSP. Our Conditional CSP subsolver (CCSP Subsolver) accomplishes this compilation task by reasoning on a Weighted Conditional, Stochastic CSP and yielding a Weighted Stochastic CSP.

Intuition: Compiling out Conditional Constraints

Here, we leverage the insight that we can augment our Weighted CSP with auxiliary variables and propositions which explicitly represent a variable’s activation status. By modifying the domains of conditional variables, creating a set of auxiliary variables, and adding constraints which explicitly represent the variable’s activation status, we are able to “compile out” the conditional constraints and create a weighted SCSP without conditionality. In Section 4.3, we will build up from this intuition to develop a procedure for compiling out conditional constraints.

3.2.2 Constraint Compiler

Finally, we are ready to go from a Weighted SCSP to a Reduced, Ordered Binary Decision Diagram! Our **Constraint Compiler** provides the parent solver with a critical part of the problem statement in Section 2.2, which established the requirement for an efficient policy representation. Rather than representing the entire explicit graph,

which can grow large when state space \mathcal{V} is large, we leverage the insight that **we can efficiently build a Reduced, Ordered Binary Decision Diagram directly from a problem’s constraints**. Our constraint compiler takes a set of constraints \mathcal{C} from our input Weighted SCSP, compiling them out using a set of efficient, canonical BDD operations, and represents the compiled constraints as a Policy BDD.

Intuition: Compiling out Conflicts

Because Binary Decision Diagrams are an efficient representation for propositional formulas and we have transformed our mixed logical and temporal problem statement into a set of unconditional, stochastic constraints, we may build a Reduced, Ordered Binary Decision Diagram (ROBDD) directly from our transformed input. **Compiling out the conflicts is as simple as using canonical BDD operations** (and, or, and not) **to incrementally build up our constraints into an ROBDD which represents the set of possible solutions to our problem**. Finally, we note that our input weights are simply weights on assignments to decision variables; because each node in a BDD is simple an assignment to a decision variable, we translate our weights \mathcal{W} from our input to their respective nodes in the Binary Decision Diagram. In Section 4.4, we will build up from this intuition to develop a procedure for compiling out conflicts into an ROBDD.

3.3 Bayesian Inference Subsolver

Our problem statement drove the requirement for a module which can efficiently use probabilistic information to infer the likelihood of an agent’s choice. Existing probabilistic information techniques, which rely on the weighting of variables according to their relative likelihoods, miss out on encoding the structural information of the problem’s probabilistic components. Such structural information enables a much more rapid inference process; **by encoding this structural information in our problem directly, we are able to speed up probabilistic inference**.

Our **Bayesian Inference subsolver** leverages insights from [11] to reframe prob-

abilistic information from a Bayesian network or its generalization, the Influence Diagram, into a set of auxiliary variables and weighted constraints which may then be added into our Weighted CSCSP for use by the Weighted CSCSP subsolver. This translation allows us to reason explicitly on the problem’s probabilistic information, encoding it directly into our ROBDD for efficient reasoning.

3.3.1 Intuition: Performing Bayesian Inference

For our Bayesian Inference subsolver, we leverage insights from [11] that note we may represent rows of a conditional probability table via a weighted set of auxiliary variables and constraints on those variables. Such insights allow us to encode probabilistic information explicitly in our policy, enabling fast online computation from the knowledge base of compiled constraints.

3.4 Temporal Subsolver

So far, we’ve discussed encoding *logical* constraints into a compact knowledge base; however, a key innovation of this thesis is the ability to mix logical constraints with temporal constraints in a chance-constrained CSP. To solve this problem, we turn to the **Temporal Subsolver**, which gathers the temporal constraints into a temporal constraint network, performs temporal reasoning, and returns a set of additional temporal constraints which denote which combinations of events will *always* be temporally infeasible as well as an implicit ordering of events derived from the temporal constraints.

3.4.1 Intuition: Extracting Temporal Conflicts

With our temporal subsolver, we leverage the insight that some assignments to decision variables will *always* yield temporally infeasible plans. We may encode these problematic assignments in our logical constraint base by performing temporal reasoning on our temporal network, extracting temporal conflicts, and representing the

conflict as a logical constraint. As an example, if the assignment of decision variable x_1 to choice c_1 always yields a temporally infeasible plan, we encode the constraint $\neq (x_1 = c_1)$ to represent that x_1 may never take the value c_1 .

Furthermore, we're interested in gleaning an *assignment order* from our temporal conflicts. To extract a variable ordering, we are able to calculate a precedence relation for the events occurring in our temporal network from graph-based algorithms. We extend this precedence relation to the choices being made at those events; from these precedence relations we may infer a variable order. Of course, our precedence relation may have ambiguity between two given events; in this case, our agent should choose a *worst-case* variable order wherein it assumes it will have to make controllable choices before observing uncontrollable observations. This pessimistic thinking ensures that the agent's estimation of *risk* will be an upper bound of the problem's actual risk. Grounding this intuition in reality, we consider the case where the temporal constraints over events $\{ev_1, ev_2, ev_3, ev_4\}$ infer the precedence relations $ev_1 \prec ev_2, ev_3 \prec ev_4$, signifying that the ordering of events ev_2 and ev_3 is ambiguous. Let ev_2 and ev_3 be associated with the uncontrollable decision H_2 and controllable decision R_3 . In our pessimistic algorithm, we want controllable assignments to *precede* uncontrollable assignments, so our desired variable ordering $\mathcal{V}_{\text{order}}$ is then given by the precedence list $ev_1 \prec ev_3 \prec ev_2 \prec ev_4$.

Chapter 4

Weighted CSCSP Subsolver

In our problem statement, we identified the need to represent conditional, stochastic problems with weights on assignments to variables as a dynamic policy. Recognizing that our chosen policy representation—the Reduced, Ordered Binary Decision Diagram—has no inherent representation of conditionality, our compilation subsolver must first eliminate the problem’s conditional variables. Once this task is completed, the compilation process may proceed, incrementally building up a policy from the problem’s set of newly-unconditional constraints. This coordination process is represented visually in Figure 4-1.

In this chapter, we describe the Weighted CSCSP solver, which compiles constraints into a policy. Building on the intuition established in 3.2, we leverage the insights that (a) we may extend our problem’s state space of variables with a set of simple compilation procedures, and (b) we may use efficient, canonical operations on a Stochastic CSP to compile our constraints into a compact policy representation.

We will begin by developing the formal problem statement for the Weighted CSCSP subsolver that meets the needs highlighted in Section 3.2, describe the high-level algorithm that coordinates between the Conditional CSP Subsolver and Constraint Compiler, and describe in detail the two modules that accomplish our goal of compiling out the problem’s expressive set of constraints.

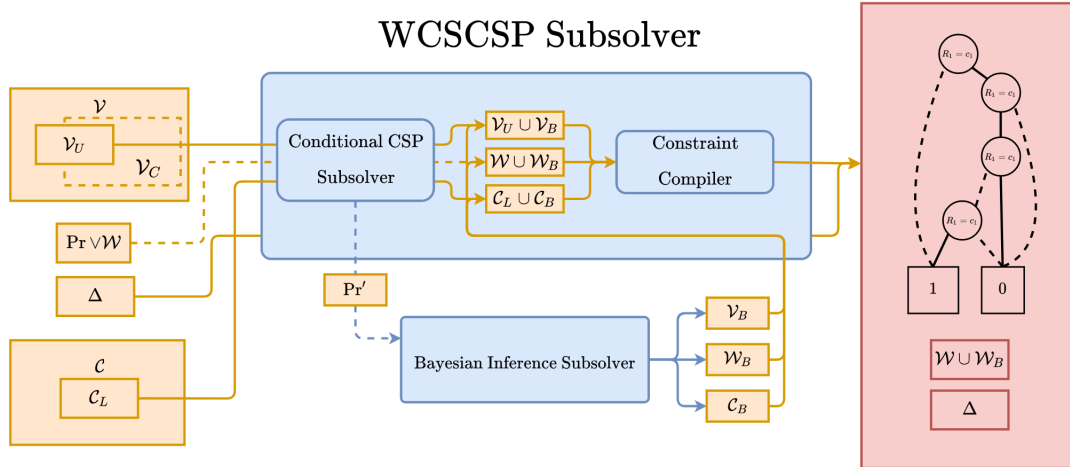


Figure 4-1: An enhanced view of the WCSCSP Subsolver architecture.

4.1 Problem Statement: Weighted CSCSPs

As described in Section 3.2, our Weighted CSCSP subsolver must accept a problem with variables that are either uncontrollable or controllable and either conditional or unconditional and a set of weights on those variables; in response, the WCSCSP subsolver should yield a Policy BDD, represented as a Reduced, Ordered Binary Decision Diagram. Here, we define our formal problem statement for the Weighted CSCSP problem. We begin by establishing the formal heritage of our input, the Weighted Conditional, Stochastic Constraint Satisfaction Problems for which our solver is named.

Definition 3. (Weighted Conditional, Stochastic Constraint Satisfaction Problem). A *Weighted Conditional, Stochastic Constraint Satisfaction Problem* (WCSCSP) extends the constraint satisfaction problem to encode assignment preferences to variables. Formally, a WCSCSP can be defined as a tuple $P = \langle \mathcal{W}, \text{CSCSP} \rangle$, where:

- \mathcal{W} , a set of weights that maps assignments of decision variables to reals \mathbb{R} . This set of weights may be arbitrarily defined over the set of decision variables, representing a set of single attribute utility functions *or* it may represent a probability distribution Pr formatted as a Bayesian network or influence diagram.
- CSCSP is a triple $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$, where:

- \mathcal{V} is the set of all the variables in the problem. Each variable $v_i \in \mathcal{V}$ is associated with a *domain* $D_{v_i} = \{d_{k_1}, \dots, d_{k_{n_1}}\}$, which describes the set of values the variable may take.
- $\mathcal{V}_U \subseteq \mathcal{V}$ is the set of stochastic variables, which have uncontrollable outcomes.
- $\mathcal{V}_{\text{active}} \subseteq \mathcal{V}$ is the subset of conditional variables that may or may not be active. Stochastic variables may be conditional.
- \mathcal{C} is the set of constraints over those variables, where each $C_i \in \mathcal{C}$ constrains the valid assignments to variables. If $v_i \in \mathcal{V}_{\text{active}}$, then all constraints C_i that contain v_i are “vacuously satisfied” when v_i is inactive.

Definition 3 combines the Conditional, Stochastic CSP from [11] with the Optimal CSP from [18]. It lends more expressiveness to the CSCSP formulation by *supporting* probabilistic constraints but not explicitly requiring them. We leverage notions from the Optimal CSP, which provides a formulation that supports a multi-attribute utility function expressed over assignments to decision variables.

Input

- A Weighted CSCSP, $P = \langle \mathcal{W}, \text{CSP} \rangle$ (Definition 3).
- $\mathcal{V}_{\text{order}}$, an ordering over the variables \mathcal{V} that specifies the order in which the variables are assigned.

Output

- A Reduced, Ordered Binary Decision Diagram (Definition 1) that represents the compiled constraints to the OCSP.
- \mathcal{W} , a set of weights on the nodes of the ROBDD.

4.2 Approach: Combining Two Solvers

Here, we describe the algorithm that coordinates between the two subsolvers (CCSP and the Constraint Compiler) as well as with the parent solver DCC-OPSAT. In short, the Weighted CSCSP subsolver first compiles out the conditional constraints, modifying the problem’s state space with an auxiliary set of variables and their constraints using the CCSP Subsolver. Note that if a probability distribution Pr is provided, this process modifies the probability distribution with the auxiliary set of variables created in the compilation process. Pr' , the modified probability distribution, is saved and distributed to the the Bayesian Inference subsolver by the parent solver.

Once Bayesian inference has been performed, the Weighted CSCSP solver can begin to compile the conditional constraints into a dynamic policy represented as an ROBDD. The constraint compilation process leverages canonical BDD operations to incrementally build up a policy from the assignments to decision variables referenced in the policy.

Algorithm 2 WCSCSP SUBSOLVER

Input: A Weighted CSCSP, $P = \langle \mathcal{W}, \text{CSP} \rangle$

Output: An ROBDD and set of weights \mathcal{W} .

- 1: **if** *the problem contains conditional constraints* **then**
 - 2: Compile out conditional constraints
 - 3: **end if**
 - 4: Hand off Pr to the Bayesian Inference subsolver
 - 5: Wait for Bayesian Inference to be performed
 - 6: Modify WCSCSP with additional probabilistic information
 - 7: Compile constraints from modified WCSCSP
-

Algorithm 2 describes the high-level procedure our Weighted CSCSP solver follows. Note the coordination with the Bayesian Inference subsolver (Chapter 5), which is responsible for turning probabilistic information modified by the Weighted CSCSP subsolver into logical constraints that may be compiled into our knowledge base of constraints.

4.3 Conditional CSP Subsolver

Recall that our problem statement may include a set of *conditional* variables that may or may not be active. While conditional variables allow a more expressive encoding, they are challenging to reason over with existing techniques. To deal with these problems, we leverage insights proposed in [11] by “compiling out” conditional variables from the CSCSP and translate our problem into a Stochastic CSP. The Conditional CSP Subsolver is responsible for the compilation of conditional variables, modification of existing probabilistic information to include the newly compiled variables, and returning a formulation with universally activated variables and constraints.

4.3.1 Problem Statement: Conditional CSPs

To better understand the CCSP Subsolver’s input, we define the Stochastic Constraint Satisfaction Problem, which encodes problems where not all assignments to variables are controllable.

Definition 4. (Weighted Stochastic Constraint Satisfaction Problem (SCSP)). A *weighted stochastic CSP (WSCSP)* extends [17] and is defined by $\langle \mathcal{V}, \mathcal{V}_U, \mathcal{C}, \mathcal{W} \rangle$. As before, \mathcal{C} is the set of constraints on variables’ domains. \mathcal{V} is an *ordered* set of variables. $\mathcal{V}_U \subseteq \mathcal{V}$ is the subset of *uncontrollable* variables whose values cannot be controlled by the agent (and, in our case, a solver). \mathcal{W} is again a set of weights on assignments to decision variables; \mathcal{W} may be either arbitrary weights or probabilistic information defined by a Bayesian network or influence diagram Pr .

Unlike the Weighted *Conditional* Stochastic Constraint Satisfaction Problem, the WSCSP does not have conditional variables or constraints. Instead, it is an *unconditional* formulation that may then be represented as a Policy BDD.

Note that traditional formulations of the SCSP [17] and CSCSP [11] include the notion of probabilistic constraints by encoding a distribution on assignments Pr and a risk bound Δ . Here, we extend this risk-bounded formulation to permit arbitrary weights on assignments to decision variables, allowing us to encode a more diverse set of problems.

Our Conditional CSP subsolver reasons on a CSCSP and compiles it into an SCSP. The output from the Conditional CSP solver may then be handed to the Bayesian Inference subsolver that, in turn, encodes the problem’s probabilistic constraints as weights on assignments to decision variables.

Input

- A Weighted CSCSP given by $\langle \mathcal{V}, \mathcal{V}_U, \mathcal{V}_{\text{active}}, \mathcal{C}, \mathcal{W} \rangle$ (Definition 3).
- A variable ordering $\mathcal{V}_{\text{order}}$.

Output

- A Weighted SCSP with conditional variables and constraints “compiled out” given by $\langle \mathcal{V}, \mathcal{V}_U, \mathcal{C}, \mathcal{W} \rangle$ (Definition 4).
- If there are conditional constraints, a modified Bayesian network Pr' , which contains the active_{v_i} constraints as outlined in Section 4.3.3.

Here, the inputs are the same as in Section 4.1; however, now instead of compiling out to a Reduced, Ordered Binary Decision Diagram we are performing the incremental step of compiling out the conditional variables.

4.3.2 Introducing Another Example

Here, we’ll introduce another example that illustrates several features of our conditional CSP subsolver. While closely related to Example 1, we introduce an additional controllable variable R_2 and mark our uncontrollable H_1 as conditional. Furthermore, we eliminate all of the temporal constraints, as they’re not relevant to the Conditional CSP subsolver and serve only to further obfuscate the relevant details.

Example 4. We introduce the simple Weighted CSCSP as follows:

- \mathcal{V} , the state space, consists of the ordered *sequence* of variables $\langle H_1, R_1, R_2 \rangle$. Each variable has the domain $\{c_1, c_2\}$.

- \mathcal{V}_U , the set of uncontrollable variables, is simply $\{H_1\}$.
- $\mathcal{V}_{\text{active}}$, the set of conditional variables, is simply $\{H_1\}$. H_1 must be activated to activate an constraints that reference H_1 .
- \mathcal{C} , the set of constraints, is simply $\{(H_1 = c_1) \Rightarrow ((R_1 = c_1) \wedge (R_2 = c_2))\}$.
- \mathcal{W} , our set of weights, simply defines the likelihood of assignments to the single uncontrollable variable H_1 :

$H_1 = c_1$	0.6
$H_1 = c_2$	0.4

4.3.3 Approach: Compiling out Conditional Variables

To go from a Conditional, Stochastic CSP to a Stochastic CSP, we *compile out* the conditional variables by introducing an auxiliary set of variables, modifying existing conditional variable’s domains, and introducing some additional constraints to reason over these conditional variables with our unconditional Policy BDD structure.

This conditional variable compilation, originally outlined in [11], is composed of three steps:

1. **Domain modification.** For every conditional variable v_i , we add the value \circ to its domain. \circ represents the v ’s *inactive value*—that is, the value the variable is assigned to when it is not active.
2. **Auxiliary active $_{v_i}$ creation.** For every conditional variable v_i , we add an auxiliary proposition active $_{v_i}$ with the domain $\{\top, \perp\}$. If v_i is active, then the proposition active $_{v_i}$ must hold.
3. **Auxiliary constraints.** A variable takes a value in its domain *if and only if* it is activated. To encode this explicitly in our SCSP, we add the constraint:

$$(\text{active}_v = \top) \Leftrightarrow \left(\bigvee_{\substack{d_i \in \text{DOMAIN}(v) \\ d_i \neq \circ}} v = d_i \right)$$

4. **Probabilistic Modifications.** If we have a probability distribution as our input Pr , we must add an additional conditional probability table that defines the active_{v_i} as the parent of conditional variable v_i . This table is specified as follows:

- For rows where v_i takes a value from its domain that is not \circ and $\text{active}_{H_1} = \top$, the probability remains the same.
- For rows where v_i takes the value \circ and $\text{active}_{H_1} = \top$, the probability is 0, representing the fact that v_i can *never* take the value \circ when it is active.
- For rows where v_i takes a value from its domain that is not \circ and $\text{active}_{H_1} = \perp$, the probability is 0, representing the fact that v_i can never take a value from its domain when it is inactive.
- For rows where v_i takes the value \circ and $\text{active}_{H_1} = \perp$, the probability is 1, representing the fact that v_i *must* take the value \circ when it is inactive.

Table 4.1 shows a concrete example if we view our example problem’s weights as encoding a Bayesian network.

Consider the conditional variable H_1 from our simple example. When H_1 is encountered by our Conditional CSP subsolver, its domain is modified from $\{c_1, c_2\}$ to $\{c_1, c_2, \circ\}$. Next, an active_{H_1} proposition is added with the domain $\{\top, \perp\}$. Finally, the auxiliary constraint from (3) is added: $\text{active}_{H_1} = \top \Leftrightarrow ((H_1 = c_1) \wedge (H_1 = c_2))$. In this way, we can explicitly represent a variable’s activation status and remove the conditionality from a given CSP.

If assignments to our conditional variable are assigned weights from \mathcal{W} , we leave them untouched and assigned the inactive value to the weight 0, thereby preserving the weighting scheme from the input Weighted CSCSP. Therefore, the outputs of our example Weighted CSCSP compilation are as follows:

1. $\mathcal{V}_C = \{\text{active}_{H_1}\}$, where active_{H_1} has the domain $\{\top, \perp\}$
2. $\mathcal{C}_C = \{\text{active}_{H_1} = \top \Leftrightarrow ((H_1 = c_1) \vee (H_1 = c_2))\}$

$H_1 = \circ$	0.0
---------------	-----

3. \mathcal{W}_C , the auxiliary weights, are given by:

As one additional detail, our auxiliary variables \mathcal{V}_B are added to the end of any existing variable order.

Note that if our weights \mathcal{W} were marked as a Bayesian network Pr, we would have to add an additional conditional probability table $\Pr(H_1|\text{active}_{H_1})$ to follow Step (4) in our procedure:

H'_1	active_{H_1}	$\Pr(H_1 \text{active}_{H_1})$
c_1	\top	0.6
c_2	\top	0.4
\circ	\top	0
c_1	\perp	0
c_2	\perp	0
\circ	\perp	1

Table 4.1: The resultant CPT added to our influence diagram

This modified probabilistic information would then be passed to the Bayesian inference subsolver and returned to the Constraint Compiler.

4.4 Constraint Compiler

The Constraint Compiler actually compiles out our constraints into dynamic policy. Recall that our policy can be represented by an explicit graph, which encodes the assignment possibilities given a history of previous assignments. By navigating down the explicit graph, following the set of observations made so far, the agent may determine which assignments do not violate the problem’s constraints, informing online decision-making. We can efficiently generate a policy by incrementally building up from decision variable assignments into a treelike data structure that efficiently represents the problem’s constraints. Our Constraint Compiler module is responsible for managing this processing, taking in a set of variables, weights, and logical constraints

and returning a Reduced, Ordered Binary Decision Diagram (ROBDD) and set of weights on the nodes of the ROBDD.

4.4.1 Problem Statement: Constraint Compiler

Recall that our input is no longer *conditional* and our probabilistic information has been encoded into an auxiliary set of variables, constraints, and weights. Of course, we are attempting to generate a policy which responds to *uncontrollable* choices, so our input remains stochastic.

Input

- A Weighted SCSP given by $\langle \mathcal{V}, \mathcal{V}_U, \mathcal{C}, \mathcal{W} \rangle$ (Definition 4). Note that \mathcal{V} is now an *ordered* set of variables, which is provided by either our problem input or the ordering inferred from the temporal constraints.

Output

- An ROBDD as defined in Definition 1.
- \mathcal{W} , a set of weights on the *nodes* in the ROBDD.

4.4.2 Simple Example: Continued

Now that we have compiled out our conditional variables, it is time to coordinate with the constraint compiler. Returning to our example, we take in the following:

- $\mathcal{V} = \mathcal{V} \cup \mathcal{V}_C$ consists of the *sequence* $\langle H_1, R_1, R_2, \text{active}_{H_1} \rangle$. The variable active_{H_1} has the domain $\{\top, \perp\}$, H_1 's domain has been augmented to be $\{c_1, c_2, \circ\}$, and R_1 and R_2 keep their original domains of $\{c_1, c_2\}$.
- \mathcal{V}_U consists again of H_1 , the lone uncontrollable variable in our example.

- $\mathcal{C} = \mathcal{C} \cup \mathcal{C}_C$ is now:

$$\begin{aligned} & \{((H_1 = c_1) \Rightarrow ((R_1 = c_1) \wedge (R_2 = c_2))\}), & (4.1) \\ \text{active}_{H_1} = \top & \Leftrightarrow ((H_1 = c_1) \vee (H_1 = c_2)) \} \end{aligned}$$

- $\mathcal{W} = \mathcal{W} \cup \mathcal{W}_C$ is now:

$H_1 = c_1$	0.6
$H_1 = c_2$	0.4
$H_1 = \circ$	0.0

Now that we have a state space of variables \mathcal{V} , a set of constraints \mathcal{C} , and an efficient variable ordering, we may begin to construct our Policy BDD using the canonical BDD operations described in Section 3.1.1. Rather than constructing a Policy BDD by building the entire explicit graph, which describes *every* possible assignment to variables, we take a more efficient conflict-directed approach and leverage the constraints themselves to build up a Policy BDD assignment-by-assignment.

4.4.3 Approach: Constructing the Policy BDD

Recall that a binary decision diagram represents a boolean formula f . Remembering that we can use the construction of an *assignment* to a decision variable to condense our arbitrarily large discrete domains in the state space \mathcal{V} to a series of binary nodes, we construct BDDs symbolically by building up our constraints from decision variable assignments to their logical equivalents. We use the operations described in [2] to perform various logical operations on any two boolean formulas f_1 and f_2 (i.e., $f_1 \wedge f_2$, $f_1 \vee f_2$, and $\neg f_1$). Algorithm 3 can be summarized as follows, and is stated more formally (as it originates in [11]) below:

1. Create decision variable assignments representing every node that appears in the constraint list \mathcal{C} .

2. Create a BDD node f_{c_i} that represents the boolean formula contained in constraint $c_i \in \mathcal{C}$.
3. Take a conjunction of all the resulting f_{c_i} , resulting in the knowledge base
$$\text{KB} \leftarrow f_{c_1} \wedge f_{c_2} \wedge \dots$$

Algorithm 3 CONSTRAINT-COMPILATION

Input: $\langle \mathcal{V}, \mathcal{C}, \mathcal{V}_{\text{order}}, \mathcal{W} \rangle$

Output: A weighted, reduced, ordered binary decision diagram KB

- 1: **for** $c_i \in \mathcal{C}$ **do**
 - 2: $N(c_i) \leftarrow []$
 - 3: **for** every decision variable assignment $v = d$ **do**
 - 4: Create a BDD node $n_i \in \mathcal{G}$ representing the assignment $v = d$
 - 5: Add n_i to $N(c_i)$
 - 6: **end for**
 - 7: Build a BDD f_{c_i} with the appropriate BDD operations on the nodes $n_i \in N(c_i)$
 - 8: $\text{KB} \leftarrow \text{KB} \wedge f_{c_i}$
 - 9: **end for**
 - 10:
 - 11: Add \mathcal{W} to the appropriate nodes in the KB **return** The BDD KB
-

Example Policy BDD

Let’s return to our example problem *after* we have compiled out the conditional constraints. Recall the relevant problem input from Section 4.4.

We’ll step through creating the policy for constraint 4.1, and show the resulting Policy BDD for all constraints in \mathcal{C} at the end. Following the recursive nature of the constraint to our first decision variable assignment, we encounter the assignment ($R_1 = c_1$) and create a node f_1 representing this assignment.

We then go to the other “innermost” constraint and encounter the assignment ($R_2 = c_2$), creating a node f_2 to represent this assignment. We create a node f_3 that is composed of the conjunction of nodes f_1 and f_2 to represent the constraint $((R_1 = c_1) \wedge (R_2 = c_2))$. Following a similar procedure, we observe that we have an implication constraint; recall that $A \Rightarrow B$ can be equivalently represented as $\neg A \vee B$. To take the negation of a BDD node, we simply swap the high and low children.

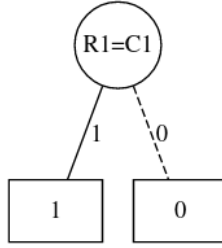


Figure 4-2: The node f_1 corresponding to the decision variable assignment ($R_1 = c_1$).

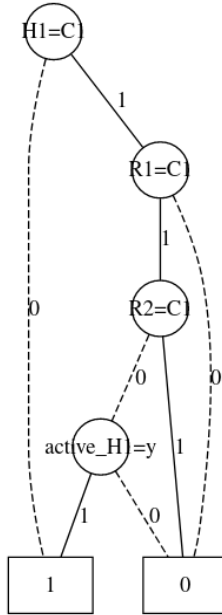


Figure 4-3: The resultant BDD node f_7 corresponding to the example constraint $\text{active}_{H_1} \Rightarrow ((H_1 = c_1) \Rightarrow ((R_1 = c_1) \wedge (R_2 = c_2)))$.

We then take the disjunction of nodes f_3 and f_4 to get a resulting node f_5 . Finally, we follow the same procedure with the node f_6 representing $\text{active}_{H_1} = \top$, connecting logical subexpressions appropriately with the node f_5 . In this way, we build up a BDD node f_7 resulting our example constraint, as showing in Figure 4-3. Recall that, of course, our variable order is $\langle H_1, R_1, R_2, \text{active}_{H_1} \rangle$, and notice that node f_7 respects this ordering.

To construct a BDD node that represents our entire knowledge base, we simply build up each constraint using the procedure described above, starting from decision variable assignments and applying the appropriate logical connections to subexpressions. Then, we create one BDD node that represents the conjunction of all the

constraint BDD nodes, as shown in Figure 4-4.

Adding Weights

Adding weights is relatively simple. Recall that our input \mathcal{W} simply maps an assignment to a decision variable; to add weights to the Policy BDD, we simply take the decision variable assignments referenced in \mathcal{W} , find the corresponding BDD node, and add the weight to the node in the Policy BDD. All other nodes receive a weight of 1. In this case, the node representing $(H_1 == \circ)$ receives a weight of 1, the nodes representing $(H_1 = c_1)$ receive a weight of 0.6, and the nodes representing $(H_1 = c_2)$ receive a weight of 0.4. Thus, we have satisfied the problem statement for our Weighted CSCSP subsolver! We went from a Weighted CSCSP and variable ordering to a Reduced, Ordered Binary Decision Diagram with weights on the nodes representing a policy. This ROBDD respects our original variable ordering.

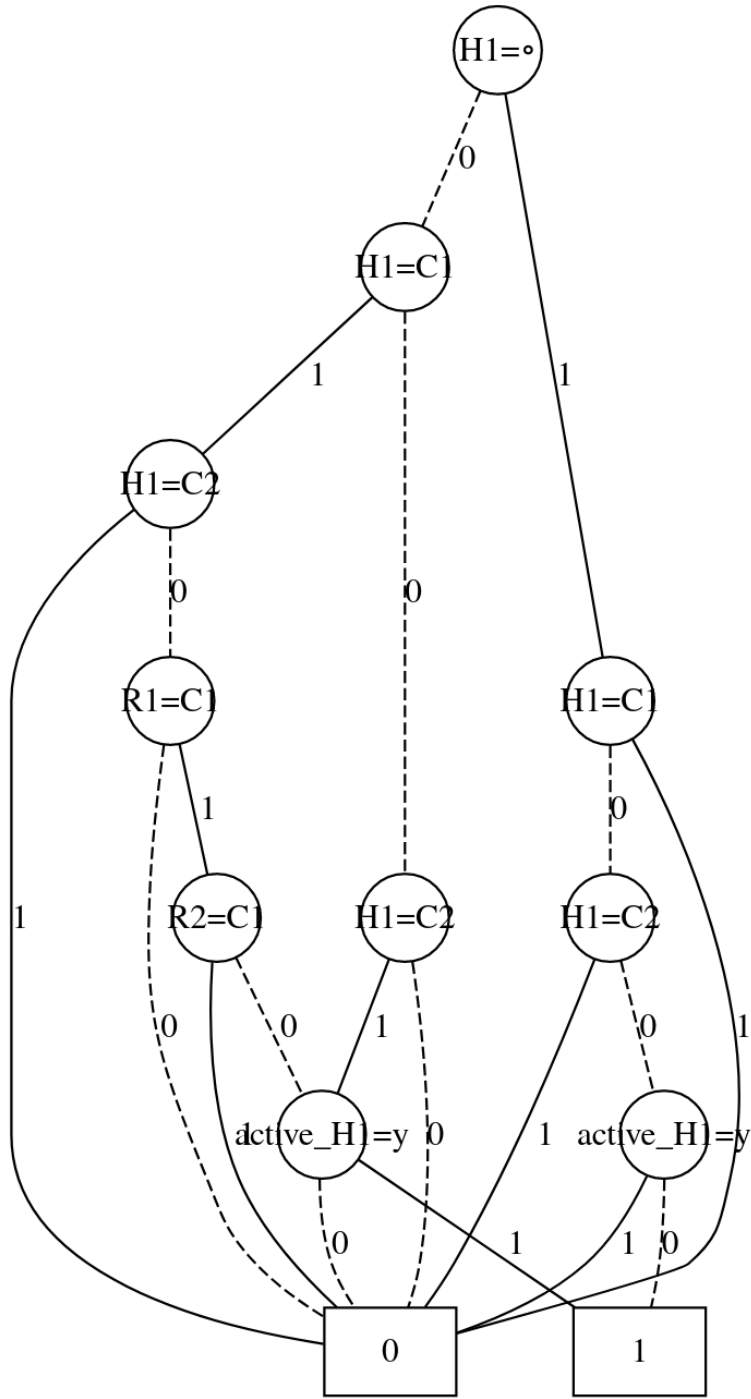


Figure 4-4: The resultant Policy BDD, which compactly encodes all of our constraints.

Chapter 5

Bayesian Network Compilation

A critical component of our problem statement was the ability to reason efficiently on probabilistic information. Our hybrid solver approaches this problem by handling the appropriate probabilistic information to a **Bayesian Inference** subsolver. Similarly to Chapter 4, we leverage the insight that by augmenting and reformulating our input with auxiliary variables and constraints, we can use existing solution methods to perform Bayesian and temporal inference. In this chapter, we'll outline the functionality and key insights of the Bayesian Inference subsolver which creates additional constraints, coordinates via the parent solver, and passes additional variables and constraints back to the constraint compiler. Then, we'll discuss in brief the approach from [11], which allows us to reason on this probabilistic information using the Policy BDD from 4 and a set of algorithms to find the set of solutions which satisfy some chance constraint online as an example of how this Bayesian information can be applied.

5.1 Problem Statement: Bayesian Inference

In this section, we'll define the formal problem statement for our Bayesian inference subsolver, which leverages techniques from [11] to efficiently represent a problem's probabilistic information as a set of additional variables and constraints.

5.1.1 Background: Probabilistic Graphical Models

Before we begin our problem statement, we discuss the requirements for our input. We leverage common probabilistic graphical models—the Bayesian Network and its generalization the Influence Diagram—to factorize our probabilistic information and develop our intuition for information. In short, a Bayesian network is a probabilistic graphical model used to represent the relative likelihoods of assignments to decision variables. Nodes in the graph represent random variables, and edges represent the conditional probability distributions over those variables. More formally, for any random variable x_i in the network with parent edges $x_{i,par_1}, x_{i,par_2}, \dots, x_{i,par_k}$, there is an associated conditional probability distribution $\Pr(x_i|x_{i,par_1}, x_{i,par_2}, \dots, x_{i,par_k})$.

The Influence Diagram [8] generalizes this notion by adding in *decision nodes* and *utility nodes*. A decision node is a node controlled by the agent; they may appear in the conditional probability table of chance nodes and therefore permit a decision node to influence its uncontrollable children. Grounding this in the human-robot collaboration domain, the robot’s decision to place a piece on the chessboard will likely impact its human partner’s future choices. Central to the influence diagram is the notion of *information arcs*, which are edges from chance to decision nodes that do *not* encode probabilistic information. Instead, an arc from node y to decision node x indicates that x will be chosen after observing the value of y .

5.1.2 Formal Problem Statement

Inputs

- \Pr , a probability distribution expressed as either a Bayesian network or influence diagram which satisfies the constraints above.
- \mathcal{V} , a reference state space used for constructing the auxiliary constraints. Because

Outputs

- \mathcal{V}_B , an auxiliary set of variables which encode the Bayesian network or influence diagram.
- \mathcal{W}_B , a set of weights on the variable domain elements $(v_i = d_j) \in D(v_i) \forall v_i \in \mathcal{V}_B$ which represent the information from the conditional probability tables.
- \mathcal{C}_B , an auxiliary set of constraints on the variables $\mathcal{V}_B \cup \mathcal{V}$ expressed in propositional state logic.

Additional Requirements on Bayesian Networks and Influence Diagrams

Our problem demands additional requirements of our input probabilistic information due to the ordering requirements, stochasticity, and conditionality encoded in our problem statement (as established in [11]). If we choose to use a Bayesian network as our Pr input, it must meet the following specifications:

- Every uncontrollable variable $v_u \in \mathcal{V}_U$ must appear as a chance node without a parent variable (representing the prior distribution of the uncontrollable variable). That is to say, an uncontrollable variable may not be probabilistically dependent on a controllable variable. If we wish to encode controllable influence, then we must use an influence diagram instead.
- The Bayesian network must respect the variable ordering; that is, a variable may not have a “parent” which appears after the variable in $\mathcal{V}_{\text{order}}$.

Similarly, if we choose to use an Influence Diagram as our Pr input, it satisfy the following constraints:

1. Every uncontrollable variable $v_u \in \mathcal{V}_U$ must appear as a chance node.
2. Every controllable variable $v \in \mathcal{V}$ that appears in Pr must be a decision node, and not all controllable variables must appear in Pr.

3. Any uncontrollable decision variable $v_u \in \mathcal{V}_U$ with a conditionally-modified domain (i.e., \circ is present in the variable’s domain) must have the auxiliary variable active_{v_u} appear as the parent decision node of v_u . The conditional probability table for $v_u \mid \text{active}_{v_u}, \dots$ must enforce the constraint that a variable takes a value from its domain when it is active (i.e., $\Pr(v_u = \circ \mid \text{active}_{v_u} = \perp, \dots) = 1$; $\Pr(v_u = d_j \mid \text{active}_{v_u} = \perp, \dots) = 0$; $\Pr(v_u = \circ \mid \text{active}_{v_u} = \top, \dots) = 0$).
4. For any uncontrollable variable in \mathcal{V}_U that precedes a controllable variable, there may not be an information arc (directed path) from the controllable variable to the uncontrollable variable. This information arc encodes variable ordering and is a concept introduced in influence diagrams. This requirement enforces the overall variable ordering.

When a Bayesian network is selected as the input and the problem contains stochastic variables, it is first compiled into the equivalent Influence Diagram satisfying the constraints above. This pre-compilation process makes it easier to reason over the problem’s probabilistic constraints and encapsulate the influence (encoded as conditional probability tables) of variables over other variables in the problem. Because Influence Diagrams explicitly reason over controllable and stochastic variables, it is easier to turn a Bayesian network when a problem is fully controllable.

5.2 Approach: Performing Bayesian Compilation

We will again leverage insights from [11] and the Weighted Model Counting (WMC) community which take a conditional probability table and turn it into additional variables and set of constraints. Such an approach allows us to encode the factored structure of a Bayesian network or influence diagram to improve inference performance. Instead of treating the probabilistic constraints as simple weights, we are able to encode the probabilistic information as a sequence of variables and set of constraints which speed up our probabilistic inference drastically. The Weighted Model

Counting (WMC) encoding procedure, which is outlined in greater detail in [11], is described in brief below.

Given an influence diagram, which is composed of a set of conditional probability tables that encode the likelihoods of decision variables, we add the following constraint to \mathcal{C}_B and \mathcal{V}_B in each row of each conditional probability table as follows:

- If the probability p associated with the row is $0 < p < 1$, we introduce a new *uncontrollable* variable θ for the row and the associated constraint (where $x_i = v_j$ denotes an assignment of variable x_i to value in its domain v_j):

$$\theta \Leftrightarrow \bigwedge_{x_i=v_j \text{ in row}} x_i = v_j$$

- If the probability p associated with the row is 0, we add the constraint

$$\neg \left(\bigwedge_{x_i=v_j \text{ in row}} x_i = v_j \right)$$

Returning to Example 1, we consider the following CPT for the prior distribution on the uncontrollable variable H_1 :

$H_1 = c_1$	0.1
$H_2 = c_2$	0.9

Table 5.1: The conditional probability table (CPT) for our simple influence diagram.

We introduce new variables to \mathcal{V}_B as shown in Table 5.2.

H_1	$\Pr(H_1)$	WMC
c_1	0.1	θ_1
c_2	0.9	θ_2

Table 5.2: The added WMC vars

These new variables will be added back into our \mathcal{V} by our parent solver when coordinating between the Bayesian inference subsolver and the Weighted CSCSP

subsolver. Similarly, we add the following set of constraints to the constraint list \mathcal{C}_B with the new variables θ_1, θ_2 from \mathcal{V}_B :

$$\theta_1 \Leftrightarrow (H_1 = c_1)$$

$$\theta_2 \Leftrightarrow (H_1 = c_2)$$

Finally, we add the following weights onto our literals as follows:

$$\mathcal{W}(H_1) = \mathcal{W}(R_1) = 1.0$$

$$\mathcal{W}(\neg H_1) = \mathcal{W}(\neg R_1) = 1.0$$

$$\mathcal{W}(\neg\theta_1) = \mathcal{W}(\neg\theta_2) = 1.0$$

$$\mathcal{W}(\theta_1) = 0.2$$

$$\mathcal{W}(\theta_2) = 0.9$$

In further constraining our problem and adding weights for the newly-defined literals, we make it much easier to reason over the probabilistic variables encoded in our problem. When returning an assignment over decision variables to a consumer, one may simply skip over the auxiliary literals to obtain a full assignment to the decision variables of the original state space \mathcal{V} .

5.2.1 A More Complex Example

Now that we've built up our intuition on a simple example, let's see what happens when we have a conditional probability table that involves a more complex set of distributions, including a conditional variable. Here, we return to Example 4. Consider the CPT 5.3 which arises from our conditional compilation procedure outlined in Section 4.3. In the far right column, we see the added WMC variables; note how

we still only add two WMC variables, θ_1 and θ_2 in order to respect the procedure outlined in Section 5.2.

H_1	active_{H_1}	$\Pr(H_1 \text{active}_{H_1})$	WMC
c_1	\top	0.6	θ_1
c_2	\top	0.4	θ_2
\circ	\top	0	
c_1	\perp	0	
c_2	\perp	0	
\circ	\perp	1	

Table 5.3: A more complex example to demonstrate our WMC encoding.

Similarly, we add the following set of constraints to \mathcal{C}_B with our auxiliary variables θ_1, θ_2 .

$$\theta_1 \Leftrightarrow (H_1 = c_1) \wedge (\text{active}_{H_1} = \top)$$

$$\theta_2 \Leftrightarrow (H_1 = c_2) \wedge (\text{active}_{H_1} = \top)$$

$$\neg((H_1 = \circ) \wedge (\text{active}_{H_1} = \top))$$

$$\neg((H_1 = c_1) \wedge (\text{active}_{H_1} = \perp))$$

$$\neg((H_1 = c_2) \wedge (\text{active}_{H_1} = \perp))$$

Finally, we add the weights onto our literals as follows:

$$\mathcal{W}(H_1) = \mathcal{W}(R_1) = \mathcal{W}(R_2) = 1.0$$

$$\mathcal{W}(\neg H_1) = \mathcal{W}(\neg R_1) = \mathcal{W}(\neg R_2) = 1.0$$

$$\mathcal{W}(\neg\theta_1) = \mathcal{W}(\neg\theta_2) = 1.0$$

$$\mathcal{W}(\theta_1) = 0.6$$

$$\mathcal{W}(\theta_2) = 0.4$$

In further constraining our problem and adding weights for the newly-defined literals, we make it much easier to reason over the probabilistic variables encoded in our problem. When returning an assignment over decision variables to a consumer, one may simply skip over the auxiliary literals to obtain a full assignment to the decision variables of the original state space \mathcal{V} .

Chapter 6

Temporal Constraint Subsolver

This thesis is motivated by the need to combine **temporal constraints** with **logical constraints**, allowing an agent to quickly infer if an assignment to finite-domain, discrete decision variables will violate a problem’s temporal constraints. In this chapter, we describe the **Temporal Constraint Subsolver**, which effectively turns temporal constraints into logical constraints via an efficient offline temporal reasoning algorithm.

In this section, we focus our attention on the **temporal reasoning** module that leverages the graph-based algorithms to infer *temporal conflicts* from a network of events and temporal constraints. The **Temporal Constraint Subsolver** gathers a set of guarded temporal constraints, events, and the controllable decision variables observed at those events, forming a labeled simple temporal network (STN) defined below in Definition 5. Then, the Temporal Constraint Subsolver uses the modified labeled APSP algorithm from Pike [11], based on the APSP algorithm from Drake [4] to generate a matrix where each entry is a Labeled Value Set (LVS) to infer a variable ordering and set of temporal conflicts. A Labeled Value Set encodes the tightest possible set of temporal constraints between two events conditioned on a set of choices. We leverage extensively from [11] as our temporal constraint subsolver inherits directly from the RIKER and PIKE executives that extract a set of temporal conflicts and variable ordering from a temporal network.

6.1 Problem Statement: Temporal Subsolver

In short, the Temporal Subsolver takes in a representation of temporal constraints, decision variables, and constraints between them to yield an auxiliary set of decision variables and constraints.

Definition 5. (Labeled Simple Temporal Network (labeled STN)) [11], [4]. A *TPN* [11] is defined by $\langle \mathcal{V}, \mathcal{E}, \mathcal{C} \rangle$.

- \mathcal{V} is set of choice variables.
- \mathcal{E} is a set of *events*, or notable points in time. Each event e is associated with a guarded simple temporal constraint in \mathcal{C} . Some events are associated with a choice variable $v_i = \text{variable} - \text{at} - \text{event}(e)$ denoting that variable v_i must be assigned by the time e is executed.
- \mathcal{C} is the set of *labeled simple temporal constraints* over \mathcal{V} . Each $c \in \mathcal{C}$ is a tuple $\langle e_s, e_f, l, u, \psi_c \rangle$, where e_s is the “start” event, e_f is the “end” event, ψ_c is a “guard” (or label) denoting a conjunction of choice variable assignments, and $l, u \in \mathfrak{R}$ represent the temporal constraint $\psi_c \Rightarrow (l \leq e_f - e_s \leq u)$. Simply put, a temporal constraint must hold if its guard is activated. Each simple temporal constraint is associated with a guard and two events. Events are executed iff they are activated (the guard $\psi_c = \top$).

Input

- A labeled STN $\langle \mathcal{V}, \mathcal{E}, \mathcal{C} \rangle$.

Output

- \mathcal{C}_{TC} , a set of inconsistent assignments that represent the conflicts extracted from temporal reasoning.
- A worst-case variable ordering $\mathcal{V}_{\text{order}}$ inferred from the problem’s temporal constraints.

6.2 Approach: The Temporal Reasoning Algorithm

Much like our Bayesian Inference subsolver, our approach is informed by the notion that we may reduce a problem into a formulation on which existing solvers may reason. We closely mirrors the approach of Drake, as employed by PIKE [11]. In short, it calculates the labeled All Pairs Shortest Path (APSP) of a labeled STN \mathcal{T} and returns a set of temporal conflicts and a variable ordering to the parent solver. High-level pseudocode for the temporal compilation procedure is shown in Algorithm 4.

Algorithm 4 TEMPORAL-REASONING

Input: A labeled STN \mathcal{T}

Output: A set of logical constraints \mathcal{C}_{TC} , auxiliary variables \mathcal{V}_{TC} , and variable $\mathcal{V}_{\text{order}}$

1: $D_{i,j} \leftarrow$ the labeled APSP of \mathcal{T}

2: $\mathcal{C}_{TC} \leftarrow$ CONFLICT-EXTRACTION($D_{i,j}$)

3: $\mathcal{V}_{\text{order}} \leftarrow$ the inferred variable ordering from $D_{i,j}$ **return** $\mathcal{C}_{TC}, \mathcal{V}_{TC}, \mathcal{V}_{\text{order}}$

6.2.1 Background: Graph-Based Temporal Reasoning

RIKER [11] proposes a modified labeled APSP algorithm, which can extract the tightest possible temporal constraints given a set of choices $V \in \mathcal{V}$. This thesis employs RIKER’s modified labeled APSP algorithm, describing in brief key concepts and how the Labeled APSP algorithm is used in our Temporal Reasoning subsolver.

Background: Labeled APSP

The Labeled APSP Algorithm, originally proposed in [4] and modified to be more efficient in [11] is a strict generalization of the Floyd Warshall algorithm. The Floyd Warshall algorithm is common for reasoning on graph-based temporal problems because it provides an efficient representation of timing constraints extracted from a weighted, directed graph called a “distance graph” [5]. The *modified Labeled APSP algorithm* takes in a labeled STN and turns it into a matrix $D_{i,j}$, where each entry

(i, j) in the matrix is the shortest path between events e_i and e_j represented as a *labeled value set* as formalized in Definition 9.

The labeled value set (LVS) was introduced in Drake’s temporal reasoning algorithms [4] and compactly records the tightest possible value for a constraint as a function of choice. To accomplish this compactness, it removes unnecessary relations and uses the notion of an *environment* (Definition 6) to efficiently represent values over many different scenarios. An LVS operates with respect to some ordering *relation* $<_R$, which provides a total ordering over elements; this relation may be strictly numerical (as in our temporal reasoning algorithms), or precedence-based (as in temporal reasoning with flexible time constraints).

Definition 6. (Environment). An *environment* φ is a partial assignment to choice variables in the state-space \mathcal{V} . In an environment, not all choice variables must be assigned. In this thesis, we denote an environment φ as a set of assignments $\{x_i = v_i, x_j = v_j, \dots\}$.

$\{\}$ represents all possible environments to the variables; logically, it corresponds to \top . In addition to using the concept of an environment, an LVS leverages the definition of a *labeled value pair*, which represents under which environments some constraint holds.

Definition 7. (Labeled Value Pair). A *labeled value pair* is a tuple (a_l, φ_l) , where a_l is some value and φ_l is an environment.

Grounding Definition 7 in an example, with the relation $<$ over some variable t , the labeled value (a_l, φ_l) means that $\varphi_l \Rightarrow t < a_l$. Notice that some constraints will be *dominated* by other constraints. For example, if we have two labeled value pairs $(2, \{\})$ and $(3, \{\})$ (encoding that t is always < 2 and t is always < 3 , respectively) over a variable t , we know that LVP $(3, \{\})$ is dominated by LVP $(2, \{\})$ because the constraint encoded by LVP $(2, \{\})$ is tighter than $t < 3$. We formalize this notion in Definition 8.

Definition 8. (Dominance). A labeled value (a_d, φ_d) *dominates* a weaker labeled value (a_w, φ_w) iff $a_d <_R a_w$ and $\varphi_w \models \varphi_d$, where \models encodes entailment.

Finally, we define a Labeled Value Set in Definition 9, which is simply a set of labeled value pairs satisfying a non-dominance constraint. All of these concepts are described in greater detail in [11].

Definition 9. (Labeled Value Set). A *labeled value set (LVS)* $L = \{(a_1, \varphi_1), (a_2, \varphi_2), \dots\}$ is a set of labeled value pairs such that no pair in the set dominates any other pair.

6.2.2 Modified Labeled APSP

Now that we've gone through some relevant background, it is time to discuss the modified APSP algorithm. The modified APSP algorithm performs the same function as the APSP algorithm, returning a matrix of labeled value sets representing the shortest path distance between pairs of events. However, in [11] it was noted that the original APSP algorithm from [4] often included labeled value sets that were not as efficient as possible. That is, they contained labeled values that were subsumed (or implied) by other labeled values based on the domain restrictions of the choice values over that the labeled value set was defined. Modified Labeled APSP improves the query quality by calculating the implications of labeled values before merging two labeled value sets, removing those that are redundant for more efficiently formulated tightest constraints.

As an example, consider the finite-domain variable x with domain $\{1, 2\}$. If we then have a labeled value set given by:

$$t < \{(3, \{x = 1\}), (4, \{x = 2\})\}$$

and we want to know the tightest constraint on t over all environment domains, we can infer a tighter bound than the one returned by the original Labeled APSP algorithm. That is, labeled APSP fails to recognize that $(x = 1) \cup (x = 2)$ actually represents all possible environments for x . Instead, Labeled APSP only determines the loosest possible time bound of $t < \infty$. However, we actually know that $t < 4$ for all possible environments! A modification that considers variables' domains allows us to extract stricter time bounds for event scheduling in our matrix D_{ij} and so represents a greater

set of temporal conflicts. By constraining our problem even more, we make it easier for our agent to reason online as it does not have to explore a large set of infeasible options.

Here, we describe the high-level procedure in Algorithm 5, describing the details intuitively but leaving detailed explanation to [11], where this procedure was originally introduced.

Algorithm 5 TEMPORAL-REASONING

Input: A labeled STN $\langle \mathcal{V}, \mathcal{E}, \mathcal{C} \rangle$

Output: $D_{i,j}$, a matrix of LVSs representing the shortest path distances between pairs of events

```

1: for each  $i, j \in \mathcal{E}$  do
2:    $D_{ij} \leftarrow \{(\infty, \{\})\}$ 
3: end for
4: for each  $i \in \mathcal{E}$  do
5:    $D_{ii} \leftarrow \{(0, \{\})\}$ 
6: end for
7: for each  $\langle i, j, l, u, \varphi \rangle \in \mathcal{C}$  do
8:    $D_{ij} \leftarrow \{(u, \varphi\}$ 
9:    $D_{ji} \leftarrow \{(-l, \varphi)\}$ 
10: end for
11: for each  $k \in \mathcal{E}$  do
12:   for each  $i \in \mathcal{E}$  do
13:     for each  $j \in \mathcal{E}$  do
14:        $C_{ij} = \text{LVSBINARYOP}(+, D_{ij}, D_{kj})$ 
15:        $D_{ij} = \text{MERGEWITHCOMPLETIONS}(D_{ij}, C_{ij})$ 
16:     end for
17:   end for
18: end for
19: return  $D$ 

```

Lines 1 through 10 of Algorithm 5 (introduced in [11]) initialize a matrix D_{ij} with a new LVS for each pair of events. In Lines 4-6, we encode the fact that the shortest distance from an event to itself can be encoded by the LVS $\{(0, \{\})\}$. Similarly, off-diagonal entry weights are added in Lines 7-10 with the corresponding lower and upper bounds for each pair of events, and all others are set to $\{(\infty, \{\})\}$. In Lines 11-18, we see the traditional triple-for-loop from Floyd-Warshall, where we iterate through double-pairs of events to improve the shortest paths between them. On Line 14, we

add two LVS’s together and then perform the MERGEWITHCOMPLETIONS procedure outlined in [11]. In short, MERGEWITHCOMPLETIONS takes in two labeled value sets and merges them into a new labeled value set leveraging the notion of a *completion*.

Let’s return to our simple example to formally describe the notion of a *completion*, which is a new labeled value set which is logically implied by others in the LVS. Consider the example from [11]:

$$t < \{(3, \{x = 1\}), (4, \{x = 2\})\}$$

where x is a finite-domain variable with domain $\{1, 2\}$.

Given this constraint, we want to know the tightest constraint on t over all environments, represented by the *query operator* $Q_L(\{\})$. Because neither $(x = 1)$ nor $(x = 2)$ is entailed by all environments, we can’t apply any bound definitively, leaving us with the loosest possible time bound $Q_L(\{\}) = \infty$. If we consider the finite domain of the variable x , however, we realize that we actually *can* learn something about the tightest possible constraint because we know that one of the two labeled values apply. Therefore, the LVS could actually be $\{(3, \{x = 1\}), (4, \{\})\}$. Such information is called a *completion*, and [11] outlines a procedure to find all of the completions of a given LVS and variable. In short, this procedure leverages insights from hyper-resolution in boolean logic to augment an LVS with new labeled value pairs that are logically implied by other labeled values, thereby changing the tightness of our query operator Q_L . By incrementally building up from the weakest possible completion $(\infty_R, \{\})$ and updated for each possible variable assignment $x = v_i$ in variable x ’s domain. This procedure *finds the completions* for a given LVS and was originally proposed in [11].

Now that we have a way of finding completions that increase the efficiency of our LVS, we need an analog for the MERGE operator proposed in [4] that leverages these completions. This is our MERGEWITHCOMPLETIONS procedure, which merges two labeled value sets and finds the valid completions. As a result of the MERGEWITHCOMPLETIONS procedure, our resultant matrix D_{ij} contains a *reduced* number of

environments in a given Labeled Value Set. Now, we have built up our intuition for creating our matrix D_{ij} using the more efficient completions representation provided by [11].

Extracting Temporal Conflicts

Recall that one of our temporal subsolver’s goals is to encode a set of *temporal conflicts* that can be extracted simply from our APSP matrix D_{ij} . Because our matrix D_{ij} encodes the *shortest possible* path between two events e_i and e_j , we know that there is a temporal conflict iff there exists some environment where an event e_i ’s shortest self-path (i.e., shortest path back to e_i) is negative. This statement has its theoretical inheritance from the concept of *negative cycles* in a distance graph from [5]. We state the formal definition of a temporal conflict in Definition 10.

Definition 10. (Temporal Conflict). An environment φ is called a *temporal conflict* iff $Q_{D_{e_i, e_i}}(\varphi) < 0$ for any event e_i .

Given a matrix D_{ij} , we need to develop a procedure to extract the full set of temporal conflicts that we may then encode as a set of logical constraints for output as \mathcal{C}_{TC} , corresponding to Line 9 of Algorithm 4. Algorithm 6 uses the fact that each entry in matrix D_{ij} is associated with an environment φ_C , which represents a partial assignment of decision variables $\{x_i, x_j, \dots\}$ to generate a logical constraint from the partial assignment. In short, if environment $\varphi_C = \{(x_i = a_i), (x_j = a_j), \dots\}$ yields a temporal infeasibility, then we add the conflict $\neg\varphi_C = \neg\{(x_i = a_i), (x_j = a_j), \dots\}$ to our set of constraints \mathcal{C}_{TC} .

Inferring Variable Ordering

We may infer the ordering of our decision variables using graph-based topological sort algorithms and the event precedence from our input labeled STN \mathcal{T} and APSP matrix D_{ij} . This procedure, introduced in [11] as `CHOOSEBESTPESSIMISTICVARIABLEORDER()` is reproduced in Algorithm 7 and described below.

Algorithm 6 CONFLICTEXTRACTION

Input: An ASPSP matrix D_{ij}

Output: \mathcal{C}_{TC} , a set of temporal constraints

- 1: $\mathcal{C}_{TC} \leftarrow \{\}$, the empty set of logical constraints
 - 2: **for each** $i \in \mathcal{E}$ **do**
 - 3: **for each** pair $(a_i, \varphi_i) \in D_{ii}$ **do**
 - 4: **if** $a_i < 0$ **then**
 - 5: $\varphi_C \leftarrow \neg\varphi_i$
 - 6: **end if**
 - 7: **end for**
 - 8: **end for**
 - 9: **return** \mathcal{C}_{TC}
-

Algorithm 7 CHOOSEBESTPESSIMISTICVARIABLEORDER

Input: An ASPSP matrix D_{ij} and TPNU \mathcal{T}

Output: $\mathcal{V}_{\text{order}}$ for every variable $v \in \mathcal{V}$

- 1: Initialize an empty graph G
 - 2: $\mathcal{V}_{\text{order}} \leftarrow []$
 - 3: **for each** $v \in \mathcal{V}$ **do**
 - 4: Add node v to graph G
 - 5: **end for**
 - 6: **for each** pair $(v_i, v_j) \in \mathcal{V}$ **do**
 - 7: **if** $Q_{D_{e_j, e_i}}(\varphi_{e_i} \wedge \varphi_{e_j} \wedge \{\}) < 0$ **then** $\triangleright e_i$ precedes e_j
 - 8: Add an edge from v_i to v_j
 - 9: **end if**
 - 10: **end for**
 - 11: **while** nodes remain in G **do**
 - 12: **if** there exists some controllable $v \in V$ with no incoming edges in G **then**
 - 13: $v_{\text{next}} \leftarrow v$
 - 14: **else**
 - 15: $v_{\text{next}} \leftarrow$ any uncontrollable $v \in V$ with no incoming edges in G
 - 16: **end if**
 - 17: Add v_{next} to the end of our $\mathcal{V}_{\text{order}}$
 - 18: Remove v_{next} and any associated edges from G
 - 19: **end while**
 - 20: **return** $\mathcal{V}_{\text{order}}$
-

We use a graph-based topological sort algorithm, inferring precedence from our APSP matrix D_{ij} as the basis for our graph. In Lines 5 through 10, we add an edge in the graph from variable v_i to v_j if the events at which decision variable v_i is observed (e_{v_i}) *precedes* event e_{v_j} . We may infer precedence from our APSP matrix D_{ij} if $D_{ij} < 0$ for the environment $\{\}$ that compactly encodes a “forall” environment, as stated more formally in Definition 11.

Definition 11. (Precedence). Event e_i with guard φ_i *precedes* event e_j with guard φ_j given context φ_C ($e_i \prec e_j | \varphi_C$) iff $Q_{D_{e_j, e_i}}(\varphi_{e_i} \wedge \varphi_{e_j} \wedge \varphi_C) < 0$. If $e_i \prec e_j$, then event e_i will be executed before e_j in all temporally consistent executions where both events are activated. If $\varphi_C \neq \{\}$, then e_i will be executed before e_j when environment φ_C holds.

Once we have inferred the precedence of events, we begin to apply our topological sorting algorithm (in this case, we borrow from [11] a modified version of Kahn’s algorithm [10] that attempts to pick controllable nodes before uncontrollable nodes to efficiently generate a pessimistic variable ordering), repeatedly selecting nodes with no incoming edges, adding it to the end of our ordering, and removing it and its edges from the graph. Kahn’s algorithm will terminate when no nodes are left in the graph G —that is, there is a topological ordering for our decision variables.

Recall from Chapter 3 how we intuited the need for a *pessimistic* variable ordering, which reasons on an upper bound of risk to create systems that act safely during their online executions. In this case, by biasing our topological sort towards controllable variables, we tend to find pessimistic variable orderings faster than the original variation of Kahn’s algorithm that selects nodes from the graph G at random.

Example 5. Let’s see what our algorithm does for a simple example given a precedence graph G given in Figure 6-1, recalling the example from Section 3.4, with events ev_1, ev_2, ev_3 , and ev_4 . Our precedence is given as follows: $ev_1 \prec ev_2$, $ev_1 \prec ev_3$, $ev_2 \prec ev_4$, $ev_3 \prec ev_4$, but we don’t know anything about the precedence relations of ev_2 and ev_3 . Each event is associated with a choice; variables starting with R are controllable (made by the robot) and variables starting with H (H_2) are uncontrollable.

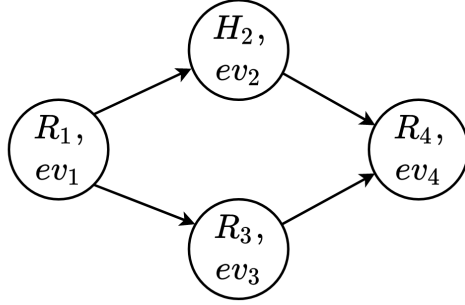


Figure 6-1: A simple precedence graph. Nodes representing decision variables are annotated with their associated events.

The precedence graph showing these relationships is given above. We first pick a random controllable variable with no incoming edges in G , per Line 12 of our Algorithm 7. R_1 is the only variable that meets this specification. We add R_1 to our variable ordering and remove it (and its associated edges) from the graph. Next, we notice that there exists another controllable $v \in \mathcal{V}$ with no incoming edges: R_3 . Picking R_3 and adding it to our variable order so $\mathcal{V}_{\text{order}} = [R_1, R_3]$, we remove all of R_3 's associated edges and continue. Now, we notice that there are *no* remaining controllable variables without incoming edges, forcing us to pick H_2 as the uncontrollable variable without incoming edges. Following the same procedure, we add H_2 and then R_4 to our variable order, resulting in a variable ordering $\mathcal{V}_{\text{order}} = [R_1, R_3, H_2, R_4]$ as desired.

The set of extracted temporal conflicts and the inferred variable ordering are enough for our temporal subsolver to pass the temporal constraints \mathcal{C}_{TC} and variable ordering $\mathcal{V}_{\text{order}}$ back to the parent solver for use by the Constraint Compiler.

Chapter 7

Discussion and Future Work

In this chapter, we will summarize the key insights from this thesis—notably, by leveraging the notion of a hybrid solver we can combine the best elements from the constraint programming and planning communities to create a solver capable of handling a rich set of problems and constraints— before highlighting some interesting open questions and directions for future work.

7.1 Discussion

In this thesis, we presented a hybrid solver dcc-OpSat that can reason on a rich set of stochastic constraints by coordinating amongst three subsolvers. In stochastic environments where agents are acting autonomously, it is essential that we develop an approach that (1) results in *risk-bounded* behavior, (2) encodes a dynamic policy conditioned on a set of observations, and (3) preserves temporal feasibility and flexibility. Most prior art solves a *subset* of these problems in isolation or is over-specified to a particular domain, resulting a constrained set of problems that the solver may handle. By coordinating between subsolvers that leverage existing techniques, this thesis allows richer expressions to be solved for efficient online reasoning.

At a high level, dcc-OpSat works by transforming its input into a set of constraint satisfaction problems with particular characteristics. By coordinating between these constraint satisfaction problems, we are able to leverage existing compilation tech-

niques to encode a policy (here, a Policy BDD) that enables fast online reasoning. Our chosen subsolvers build on key insights from the knowledge compilation, SAT, and temporal reasoning communities, providing us with a way to reason on problems with *probabilistic information, temporal constraints, action models, conditionality, risk, and uncertainty*. In short, dcc-OpSat is a highly flexible architecture that combines the best aspects from much prior work to solve dynamic, chance-constrained CSPs with mixed logical and temporal constraints.

7.2 Future Work

There are a number of interesting avenues for future work that build on the hybrid solver proposed in this thesis! We discuss a few here.

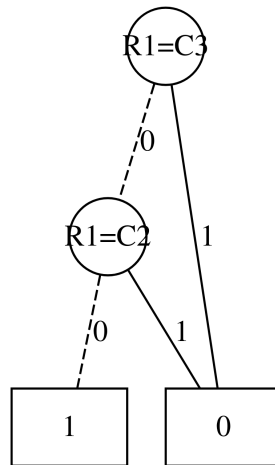


Figure 7-1: An alternative encoding of the constraint $(R_1 = c_1)$.

While the Policy BDD is an excellent representation of *binary* variables, it is not as efficient for variables with more than two values in their domain. As an example, if a variable R_1 has the domain $\{c_1, c_2, c_3\}$, encoding the constraint $(R_1 = c_1)$ may be equivalently represented as $\neg(R_1 = c_2) \wedge \neg(R_1 = c_3)$ as shown in Figure 7-1. Because we do not consider ordering over decision variable assignments, only decision variables, and we know that BDD size efficiency is dramatically impacted by the ordering of its nodes, there are possibilities for making this Policy BDD representation even more efficient when considering variables with non-binary domains. Another

alternative would be considering a different encoding of the policy entirely (for example, Multivalued Decision Diagrams [3]), or providing the hybrid solver with a more expressive and generic policy representation so that a user may pick their chosen alternative that specializes on these alternatives.

Additionally, our uncertainty comes from decision variables, not temporal constraints. While a useful representation for the set of problems described above, an interesting extension would be one that allows *contingent* temporal constraints, or temporal constraints that have uncontrollable durations. Grounding this notion in reality, if the time that the human took to do the rivets depended on whether or not they were using a battery-powered rivet gun or one connected to an outlet. It is possible that we could leverage insights from this work to encode the flexible duration as a function of some decision variable; alternatively, we could represent the temporal uncertainty as a probability distribution and reason about the controllability of the schedule using the concepts from STNUs [14] or pSTNs [16]. A pSTN would likely be the most natural extension as we already support probabilistic information in the form of Bayesian networks and Influence Diagrams; encoding temporal uncertainty probabilistically would be a fascinating extension to this work.

Finally, it would be interesting to investigate avenues in which a robot takes actions to observe when it is uncertain, rather than just waiting for the problem's risk to decrease. It is possible to leverage insights from [1] to increase our human-robot's likelihood of success through pointed, useful questions which allow our agent to reason on its environment. By folding the innovations from [1] into this hybrid architecture, we may create a more capable agent capable of resolving uncertainty.

Appendix A

Extensions

In this chapter, we discuss some extensions to our hybrid solver that make it useful in planning problems. First, we’ll talk about how to reason on a Policy BDD, extracting a best policy from the explicit graph using the BDD-MAX Σ II algorithm introduced in [11]. Then, we’ll discuss some extensions to our temporal reasoning algorithms that allow us to encode activities with preconditions and effects, reasoning over the *causal links* to determine which activities must occur in order to have a successful execution.

A.1 Extension: Reasoning on a Policy BDD

First, we discuss briefly a set of algorithms which allow us to determine the “best” future assignment given a history of assignments to decision variables. These algorithms are not a core component of our hybrid solver. Instead, they demonstrate one use case for flexibly human-robot collaboration which our hybrid solver supports and enables. This approach is directly informed by [11], who designed these algorithms to enable risk-bounded collaboration in a human-robot team. This thesis references them only as one *application* of dcc-OpSat to human-robot collaboration.

In this section, we’ll build up the intuition for extracting the best policy from a Policy BDD, starting from an explicit graph without the the added WMC constraints, adding in the WMC constraints, and finally extending the concepts to a Policy BDD. Querying the Policy BDD to find the best policy enables a consumer of the BDD

to determine *which* choice is the best given a partial assignment to controllable and uncontrollable decision variables and *how* to act optimally in the future. Such algorithms are useful for online execution, whereby an executive is able to quickly reason over its observations and make choices to geared towards success.

A.1.1 Background: Extracting the Best Policy from Explicit Graphs

Recall that our BDD performs two tasks: first, encoding all possible candidate solutions to a conditional, stochastic CSP. Second, it allows us to reason quickly online to determine *if* there exists some full assignment given a partial assignment to variables that satisfies a chance constraint and *which* full assignment has the greatest likelihood of success. We will build up to the BDD-MAX $\Sigma\Pi$ algorithm proposed in [11] by first discussing how to extract the best policy from a graph *without* the structural encoding of the probability distribution described in Section 5.2. Then, we will demonstrate that correct probabilistic information can be extracted from the explicit graph with the auxiliary Weighted Model Counting variables.

Policy Extraction *Without* WMC Constraints

The intuition for generating the weight β_n to node n is rooted in the theoretical basis of the *expectimax* or *average-out and fold-back* algorithms developed by the stochastic game search community ([17], [9], [12]) for search over explicit graphs and extended to Policy BDDs in [11]. To review, we'll examine the *expectimax* algorithm in the context of an explicit graph over a sequence of uncontrollable and controllable assignments to decision variables.

Recall the simple explicit graph introduced in Section 3.1.1 and annotated with the resultant β_n from following the described algorithm, as shown in Figure A-1. Note that the values at the leaf nodes are generated arbitrarily for pedagogical purposes, and their generation is not described within this thesis.

We traverse the graph from leaf to root, working bottom-up to store a value β_n in

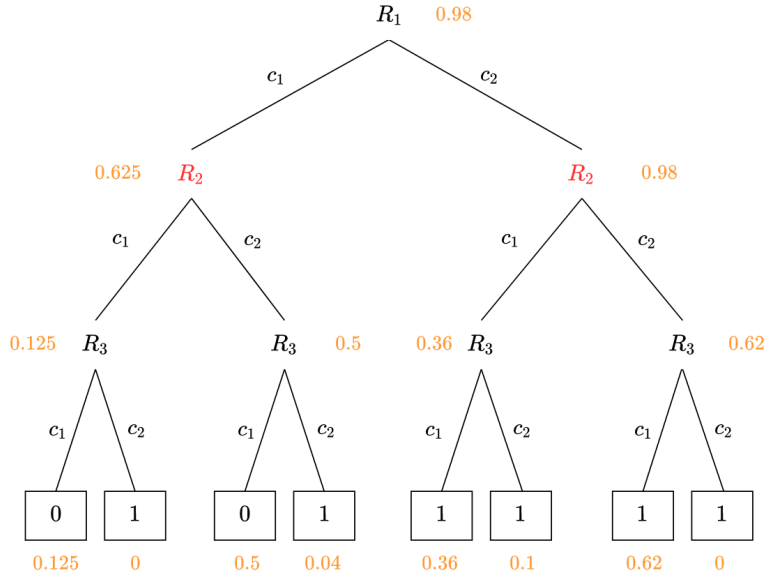


Figure A-1: A simple explicit graph, with a uniform prior distribution..

node n . For a leaf node ψ_s in the explicit graph, $\beta_n = \Pr(\psi_s)\mathbf{1}_{sat}(\psi_s)$, which represents the joint probability of ψ_s if interpretation ψ_s satisfies the problem's constraints and 0 otherwise (the path terminates in the 0 value). Moving up to the node's parents, we assign the parent value based on its variable type:

$$\beta_n = \begin{cases} \max_{n_i \text{ child of } n} \{\beta_{n_i}\} & v \text{ is controllable} \\ \sum_{n_i \text{ child of } n} \beta_{n_i} & v \text{ is uncontrollable} \end{cases}$$

As the process moves up the tree, it takes the appropriate maximum or sum; if taking the max, the algorithm records the node that resulted in the maximum value. Once the root node n_{root} is reached, the optimal policy π^* is extracted by tracing down the tree along the children that yielded the best values. Note that the probability of satisfying the constraints for the optimal policy π^* is conveniently represented as $\beta_{n_{root}}$, making it efficient to determine if there is a policy that satisfies the chance constraint.

Policy Extraction *With* WMC Constraints

In Section 5.1.2, we described the procedure by which we encode probabilistic information explicitly in the policy of a Conditional, Stochastic CSP. Now, we'll use the weights on the θ_i variables to calculate the β_n for nodes in an explicit graph with the WMC variables included using the COMPUTEBETATREE algorithm originally proposed in [11] and reproduced below in Algorithm 8 for convenience.

Algorithm 8 COMPUTEBETATREE

Input: A node n representing variable v

Output: The value β_n

- 1: **if** n is a terminal node **then**
 - 2: $\beta_n \leftarrow \begin{cases} 0 & \text{if } n \text{ is terminal } 0 \\ 1 & \text{if } n \text{ is terminal } 1 \end{cases}$
 - 3: **else**
 - 4: $\beta_{n_i}, \dots \leftarrow \text{COMPUTEBETATREE}(n_i)$ for each child n_i of n corresponding to assignment $v = d_i$
 - 5: $\beta_n = \begin{cases} \max_{n_i \text{ child of } n} \{\beta_{n_i}\} & v \text{ is controllable} \\ \sum_{n_i \text{ child of } n} \mathcal{W}(v = d_i)\beta_{n_i} & v \text{ is uncontrollable} \end{cases}$
 - 6: **end if**
 - 7: **return** β_n
-

Algorithm 8 is remarkably similar to the approach for finding the best policy *without* the WMC constraints. This new method, however, allows us to reason over an influence diagram that captures relationships between variables, rather than just reasoning on the prior distribution of the leaf nodes. Recall that we consider the θ_i variables to be uncontrollable; by chaining the θ_i variables together, Algorithm 8 allows us to correctly compute the joint probability of their associated nodes, as demonstrated and described more fully in [11].

A.1.2 Extracting the Best Policy from the Policy BDD

Now that we have the background of policy extraction from an explicit graph, we extend policy extraction to Policy BDDs using the BDD-MAXΣΠ algorithm developed in [11]. While the intuition is similar to Algorithm 8, there are two key considerations

for operating on a BDD with WMC constraints explicitly encoded. First, **variables may be skipped along a path from root to leaf** if either value for those variables could be chosen. Therefore, *we must consider all the possible assignments along paths with variables omitted*. Second, **BDD nodes may be re-used** due to the compact and efficient structure of BDDs.

As with the explicit graph, we associate each node in the Policy BDD with a value β_n , which again represents the likelihood of success if the agent acts optimally from that point on. As before β_n is computed recursively from the children of node n and cached at each node; this caching allows us to avoid re-computation, as BDD nodes should be immutable and so their β_n values will never changed.

[11] introduces the concept of a *path factor* to account for skipped variables in a traversal of a Policy BDD. In short, if two subtrees of the explicit graph are identical, the redundant substructure is collapsed in the Binary Decision Diagram, and the variable is skipped. The BDD-MAX $\Sigma\Pi$ algorithm is described in great detail in [11, p. 266]; here, we simply use the results to evaluate the likelihood of success given the partial assignment to a set of decision variables.

A.1.3 Using the Policy BDD in Online Execution

An *executive* is generally responsible for scheduling and dispatching an agent’s activities and monitor the execution to determine if the agent will fails or has already failed to execute a problem correctly, subject to some constraints. Central to the success of a dynamic executive is the ability to reason quickly about the choices made by the agent *and* its environment. The OpSat CSP solver is used by executives to quickly, but lazily, compile out constraints and find a single *optimal* set of decisions using Conflict-Directed A* search [18].

A.2 Extension: Solving a Planning Problem

Here, we discuss how a planner might interact with dcc-OpSat, using its action model to represent the notion of causality which can be encoded as a set of additional tem-

poral constraints for our hybrid solver, dcc-OpSat. This extension demonstrates one application of the dcc-OpSat solver, which allows us to solve dynamic human-robot collaboration problems. Consider the human-robot collaboration example, where a human and robot are working together to prepare the human for a busy workday. If the robot does not pump up the bike’s tires, then the human cannot ride their bike to work. This simple demonstration suggests the notion of *labeled causal links* and *producer* and *consumer* events. The theoretical inheritance of these structures and algorithms are described in great detail in [11]; here, we simply examine the input and output and describe the algorithm’s functionalities in brief.

We extend the algorithm and problem statement to accept a Temporal Plan Network under Uncertainty (TPNU) rather than a labeled STN and an *action model*, which encodes actions precondition’s and effects. As an example, the *effect* of the pump tires actions would be that the bicycle’s tires are pumped. Additionally, we modify our temporal reasoning algorithm to run *twice*: first to extract the labeled causal links which can encode additional temporal constraints, generating a second TPNU \mathcal{T}' , and the second time to extract temporal conflicts from the modified TPNU. These modifications demonstrate how our hybrid solver can be applied to solve planning problems; because the formulation is so flexible, it allows for the encoding of additional extensions.

Definition 12. (Temporal Plan Network (TPN)) [11], [4]. A *TPN* [11] is defined by $\langle \mathcal{V}, \mathcal{E}, \mathcal{C}, \mathcal{A} \rangle$.

- \mathcal{V} is set of choice variables.
- \mathcal{E} is a set of *events*, or notable points in time. Each event e is associated with a guarded simple temporal constraint in \mathcal{C} . Some events are associated with a choice variable $v_i = \text{variable} - \text{at} - \text{event}(e)$ denoting that variable v_i must be assigned by the time e is executed.
- \mathcal{C} is the set of *guarded simple temporal constraints* over \mathcal{V} . Each $c \in \mathcal{C}$ is a tuple $\langle e_s, e_f, l, u, \psi_c \rangle$, where e_s is the “start” event, e_f is the “end” event, ψ_c is

a “guard” denoting a conjunction of choice variable assignments, and $l, u \in \mathfrak{R}$ represent the temporal constraint $\psi_c \Rightarrow (l \leq e_f - e_s \leq u)$. Simply put, a temporal constraint must hold if its guard is activated. Each simple temporal constraint is associated with a guard and two events. Events are executed iff they are activated (the guard $\psi_c = \top$).

- \mathcal{A} is a set of *activities*, which denote the actions that must be executed during a particular temporal constraint. Stated formally, each activity $a \in \mathcal{A}$ is a tuple $\langle c, \alpha \rangle$, where c is a temporal constraint and a is an action that must be executed online. Action α starts when e_s is scheduled and ends when e_f is scheduled.

Input

- A TPNU $\langle \mathcal{V}, \mathcal{E}, \mathcal{C}, \mathcal{A} \rangle$.
- An action model which specifies the preconditions and effects for each activity $a \in \mathcal{A}$.

Output

- \mathcal{C}_{TC} , a set of additional propositional constraints which represent the conflicts extracted from temporal reasoning.
- \mathcal{V}_{TC} , variables representing producer and consumer events **if** an action model \mathcal{A} is provided in the input.
- A worst-case variable ordering $\mathcal{V}_{\text{order}}$ inferred from the problem’s temporal constraints.

Before getting started with the action model portion of our problem, it is helpful to define the concept of a **producer event**, a **consumer event**, and a **labeled causal link** as outlined in [11].

Definition 13. (Producer and Consumer Event). A *producer event* is an event e_p which has an effect p that is a precondition of the **consumer event** e_c .

Algorithm 9 MODIFIED-TEMPORAL-REASONING

Input: A TPNU \mathcal{T} and an optional action model \mathcal{A}

Output: A set of logical constraints \mathcal{C}_{TC} , auxiliary variables \mathcal{V}_{TC} , and variable $\mathcal{V}_{\text{order}}$

- 1: $D_{i,j} \leftarrow$ the labeled APSP of \mathcal{T}
 - 2: Extract the labeled causal links from \mathcal{T} using $D_{i,j}$
 - 3: Generate an augmented TPNU \mathcal{T}' using \mathcal{T} and labeled causal links
 - 4: $\mathcal{V}_{TC} \leftarrow$ the added producer and consumer events
 - 5: $\mathcal{C}_{TC} \leftarrow$ the conflicts extracted from the labeled causal links
 - 6: **if** any temporal constraints were added to \mathcal{T}' **then**
 - 7: $D_{i,j} \leftarrow$ the labeled APSP of \mathcal{T}'
 - 8: **end if**
 - 9: $\mathcal{C}_{TC} \leftarrow$ CONFLICT-EXTRACTION($D_{i,j}$)
 - 10: $\mathcal{V}_{\text{order}} \leftarrow$ the inferred variable ordering from $D_{i,j}$ **return** $\mathcal{C}_{TC}, \mathcal{V}_{TC}, \mathcal{V}_{\text{order}}$
-

Grounding our definition back in our example, the producer event would be the event where the robot pumps the tires, and the consumer event is the one where the human begins riding their bike to work.

Generating \mathcal{T}' from Causal Links

Given a totally ordered plan and action model, inferring causal links is straightforward; we simply work backwards and find the latest producer event for a given consumer event. We then know that there is a *causal link* between the latest producer and the consumer event; we may encode this causal link as an additional temporal constraint. In general, though, this is challenging for partially-ordered plans with metric temporal constraints. Consider two producer actions whose associated events do *not* have a strict temporal precedence; in this case, it is unclear which producer is the “later” producer and therefore relevant to the causal link structure. As a result, there may be multiple possible candidate causal links. Additionally, a key part of our problem statement was the notion of *choice* — that is, the activation of certain producer events is conditioned upon choices made by the agent and its environment. To solve this problem, we replicate the approach produced in [11], which introduces the notion of *labeled causal links*, whereby a traditional causal link is labeled with the choices that guard the activation of its events.

Definition 14. (Labeled Causal Link) [11]. A *labeled causal link* is a tuple $\langle e_P, e_c, p, \varphi \rangle$, where e_P is the *producer event* with $p \in \text{EFFECTS}(e_P)$, e_c is a consumer event occurring after e_P with $p \in \text{PRECONDITIONS}(e_c)$, p is a predicate, and the label ψ is an *execution environment* e_P, φ_{e_P} .

We can also generalize the notion of a *threat*, where an action’s effects negate the preconditions of another action. While an agent can never work to remedy the inactivation of its latest causal link, it *can* react to a threat by taking an action whose effects undo the threat actions’ effects. Stated more formally, if a threat e_t has $\neg p$ as its effects, and is activated (and the related consumer and producer are activated), the executive could either deactivate the consumer or try to produce the effect with another action later in the execution.

Finally, we define the notion of the *dominance* of causal links, which simply says that some causal links are irrelevant because they will always be subsumed by other causal links.

Definition 15. (Labeled Causal Link Dominance) [11]. A labeled causal link $\langle e_{P_i}, e_c, p, \varphi_{e_{P_i}} \rangle$ *dominates* another labeled causal link $\langle e_{P_j}, e_c, p, \varphi_{e_{P_j}} \rangle$ iff:

- $\varphi_{e_{P_i}} \prec e_c |_{\varphi_{e_{P_j}}}$
- $\varphi_{e_{P_j}} \models \varphi_{e_{P_i}}$
- $\varphi_{e_{P_j}} \prec e_{P_i} |_{\varphi_{e_c}}$

Because these causal links are dominated, we can keep our augmented TPNU \mathcal{T}' small by pruning them as their dominance is discovered. In this case, we leverage insights from [11] to extract these causal links, treating their results as a given from a black box and describing in detail the logical encoding of causality, producers, and consumers.

In short, we use the labeled causal links to encode additional temporal constraints and logical constraints in our problem. We then run the Modified APSP algorithm described in Section 6.2.2 on the modified temporal network, extracting a new set of temporal conflicts to be added to \mathcal{V}_{TC} . We describe the procedure intuitively,

beginning with determining which producers support the precondition p of some event e_c . As is a theme, we add an auxiliary variable which encodes which of the different possible producer events will enforce the support for precondition p of event e_c ; that is to say, which event will be the producer in the associated causal link. This new variable s_{p,e_c} choosing between, as an example, two events e_{P_1} and e_{P_2} is defined as follows:

$$s_{p,e_c} = \begin{cases} e_{P_1} & \text{if } e_{P_1} \text{ will be the producer} \\ e_{P_2} & \text{if } e_{P_2} \text{ will be the producer} \\ \circ & \text{if neither will be the producer} \end{cases}$$

Note that s_{p,e_c} is a *conditional* variable; it may be inactivated if e_c is inactive. Therefore, we must also add a variable $\text{active}_{s_{p,e_c}}$ to respect our conditional formulation from Section 4.3.

If there is a threat, we define the controllable variable $o_{p,e_c,e_{P_1},e_{T_1}}$ where e_{P_1} is the producer event and e_{T_1} is a threat event as follows:

$$o_{p,e_c,e_{P_1},e_{T_1}} = \begin{cases} -1 & \text{if } e_{T_1} \text{ precedes } e_{P_1} \\ +1 & \text{if } e_{T_1} \text{ succeeds } e_{P_1} \end{cases}$$

Note that, again, $o_{p,e_c,e_{P_1},e_{T_1}}$ is *conditional* and may not be active.

Finally, we add a set of additional logical constraints encoding these causal links and temporal constraints to generate our augmented TPNU \mathcal{T}' as shown in Table A.1.

Once we have added our auxiliary logical and temporal constraints, we re-run labeled APSP on our new TPNU \mathcal{T}' to generate an additional set of temporal conflicts \mathcal{C}'_{TC} . Finally, we return the complete set of temporal conflicts $\mathcal{C}_{TC} \cup \mathcal{C}'_{TC}$, the additional variables \mathcal{V}_{TC} , and logical constraints \mathcal{C}_{TC} to our parent solver. Now, our Temporal Subsolver has completed its reasoning!

Constraint	Reason
Propositional for \mathcal{C}_{CT} :	
$\varphi_{e_c} \Rightarrow \bigwedge_{e_{P_i} \in P} (s_{p,e_c} = e_{P_i} \wedge \varphi_{e_{P_i}})$	At least one activated producer
$\neg \varphi_C$	Avoid definite threat
$\neg((s_{p,e_c} = e_P) \wedge \varphi_{e_c} \wedge \varphi_{e_P} \wedge \varphi_{e_T})$	Avoid definite threat
$\neg \varphi_C$	Avoid temporal conflict
$\text{active}_{s_{p,e_c}} \Rightarrow \bigvee_i ((s_{p,e_c} = e_{P_i}) \wedge \varphi_{e_{P_i}})$	Effect is produced and maintained
$\varphi_{e_c} \Leftrightarrow \text{active}_{s_{p,e_c}}$	s_{p,e_c} is activated when e_c is executed
$\varphi_{e_v} \Leftrightarrow \text{active}_v$	A variable must be assigned when the event it is associated with will execute
$(s_{p,e_c} = e_P) \wedge \varphi_{e_c} \wedge \varphi_{e_P} \wedge \varphi_{e_T} \Leftrightarrow \text{active}_{o_{p,e_c,e_{P_1},e_{T_1}}}$	Threat conditionality
Temporal for \mathcal{T}'	
$e_{P_i} \xrightarrow{[\epsilon, \infty]} e_c : \{s_{p,e_c} = e_{P_i}\} \wedge \varphi_{e_{P_i}} \wedge \varphi_{e_c}$	Producers precede consumers
$e_c \xrightarrow{[\epsilon, \infty]} e_{T_j} : \varphi_C$	Force threat after consumer
$e_{T_j} \xrightarrow{[\epsilon, \infty]} e_{P_i} : \varphi_C$	Force threat before producer
$e_{T_j} \xrightarrow{[\epsilon, \infty]} e_{P_i} : \{p_{p,e_c,e_{P_i},e_{T_j}} = -1\} \wedge \varphi_C$	Force threat before producer
$e_c \xrightarrow{[\epsilon, \infty]} e_{T_j} : \{p_{p,e_c,e_{P_i},e_{T_j}} = +1\} \wedge \varphi_C$	Force threat after consumer

Table A.1: The additional propositional constraints for \mathcal{V}_{CT} and \mathcal{T}'

Appendix B

Implementation

Here, we'll talk about how to actually *use* DCC-OPSAT, translating a mathematical problem statement into a CSP represented in MiniZinc [15]. First, we'll talk about the systems involved: Odo (our CSP modeling language), OpSat, and RIKER. Then, we will explicitly describe the inputs and outputs, examining the extracted temporal conflicts, auxiliary, constraints, and resultant ROBDD. Finally, we'll discuss a few of the implementation details which were not central to the theoretical background of this thesis but are useful to anyone who wishes to continue this work.

B.1 Systems Used

In this section, we introduce the systems used to solve this problem.

B.1.1 Odo

Odo provides an end user with extensible, flexible CSP modeling capabilities and the ability to translate from the MiniZinc modeling language into a formulation accepted by our CSP solvers. In this thesis, we focus on the existing API for **CSPs**, which are in turn composed of a **state space** of **variables** and set of **constraints** on those variables. Our architecture extends Odo with the following representations necessary to fully model the CSCSP and pTPN problem statements:

1. **Bayesian Networks and Influence Diagrams.** In order to encode probabilistic information about the problems, it was necessary to provide a user with a way of modeling Bayesian Networks and their generalization, Influence Diagrams. We based the Odo API on the RIKER implementation, allowing a user to parse Bayes nets and Influence Diagrams from their respective open standards (XMLBIF and PGMX) and represent them in a way the subsolvers can understand.
2. **Policy BDD.** A problem’s constraints and state space are compiled into a Policy BDD. Odo’s modeling language is augmented to provide a user with a standard API for interacting with a Policy BDD, allowing them to query it for solutions and determine if a given set of decisions yields a solvable sub-problem.

B.1.2 Riker

RIKER is the theoretical backbone of our hybrid architecture. Originally proposed (and implemented) in [11], it is capable of taking in a pTPN, Bayesian network, and PDDL action model to produce a Policy BDD which reasons on the pTPN’s implicit constraints. RIKER also interfaces with the PIKE executive, operating on a Policy BDD to greedily make choices based on a history of assignments to a problem’s decision variables. This work leverages the following subcomponents of RIKER:

1. **CSCSP Modeling Language.** RIKER implements a CSCSP modeling language, which is capable of representing a state space of variables in an efficient, hash-table-based memory structure. Furthermore, it can model *assignments* to those decision variables and reason on them quickly online.
2. **Bayesian Network Parsing and Implementation.** RIKER provides a user with libraries for parsing and modeling Bayesian Networks (and their generalizations, Influence Diagrams). The hybrid architecture proposed in this thesis uses these parsing libraries for convenience and bases the extensions to the Odo modeling language loosely on the RIKER Bayesian Network implementation included in RIKER.

3. **Policy BDDs.** RIKER uses a Policy BDD (among other representations) to store its knowledge base of constraints, which represents the fully compiled set of constraints. Rather than reinvent the wheel with a novel policy representation, our hybrid architecture leverages the Policy BDD structure and several RIKER-specific implementation details.
4. **Temporal Reasoning.** RIKER is capable of extracting a Conditional, Stochastic CSP from a probabilistic Temporal Plan Network using the techniques outlined in [11]. We leverage this infrastructure for our Temporal Constraint solver (Section 3.4).

B.1.3 OpSat-v3

OpSat-v3 is a Conflict-Directed A* search CSP solver. Traditionally, OpSat has taken in a CSP (composed of variables and constraints) and returned the *single optimal solution* by searching over the state space of variables and propagating constraints to prune invalid portions of the search tree. Central to this search is the notion of *lazily* compiling constraints as they are relevant. In this thesis, we extend OpSat’s architecture to compile out *all* of the constraints online to create a Policy BDD when an input CSP contains uncontrollable variables. In this way, we augment an optimal satisfiability solver to solve a broader set of problems more useful for domains involving human-robot collaboration.

B.2 Our Simple Example

Recall Example 1. We reproduce the problem here and explicitly describe the full set of logical and temporal constraints.

B.2.1 Problem Statement

Our state space consists of the following:

The temporal constraints \mathcal{C}_T are given Table B.2.

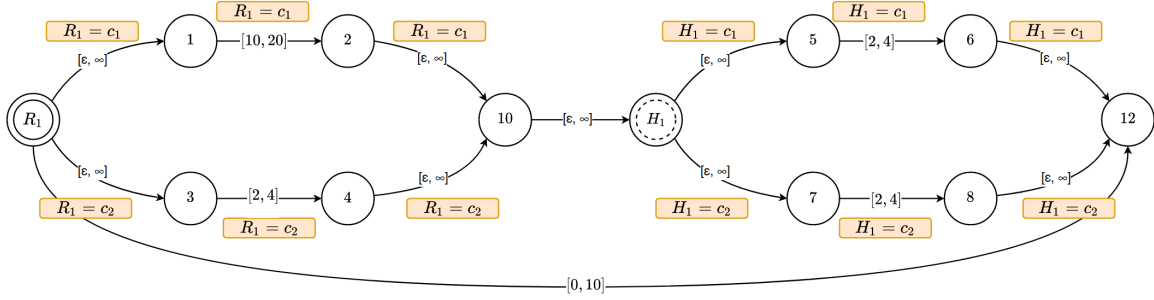


Figure B-1: A visual representation of our mixed-logic temporal CSP. Edges denote *temporal constraints* with of the form $[\text{lower bound}, \text{upper bound}]$, denoting the *temporal distance* between events. Controllable choices are marked by a double circle, with uncontrollable choices marked by a dashed circle.

Variable	Domain
Logical	
R_1	$\{c_1, c_2\}$
H_1	$\{c_1, c_2\}$
Temporal	
$ev_9, ev_1, ev_2, ev_3, ev_4,$ $ev_{10} ev_5, ev_6, ev_7, ev_8, ev_{12}$	$[-\infty, \infty]$

Table B.1: State space for our example problem.

The logical constraints \mathcal{C}_L are given simply by $\{(H_1 = c_1) \Rightarrow (R_1 = c_1)\}$. Finally, our influence diagram is given by Table B.3

B.2.2 Extracted Temporal Conflicts

Temporal reasoning extracts the conflict $\neg(R_1 = c_1)$. This is because setting $(R_1 = c_1)$ will enforce that we have a minimum temporal distance between events 9 (our start event) and 12 (our end event) of 12 ($\llbracket [10, 20] \rrbracket + \llbracket [2, 4] \rrbracket = 12$). Recall that we have a time constraint between events 9 and 12 stating that our overall execution time may not exceed 10; therefore, we are left with a temporal conflict! Our temporal reasoning algorithm correctly extracts this conflict and adds it to the knowledge base of conflicts. Finally, we may also extract the variable ordering $\langle R_1, H_1 \rangle$ following the variable ordering procedure outlined in Algorithm 7.

Start Event	End Event	STC	Guard
ev_9	ev_1	$[0, \infty]$	$R_1 = c_1$
ev_1	ev_2	$[10, 20]$	$R_1 = c_1$
ev_9	ev_{10}	$[0, \infty]$	$R_1 = c_1$
ev_9	ev_3	$[0, \infty]$	$R_1 = c_2$
ev_3	ev_4	$[2, 4]$	$R_1 = c_2$
ev_4	ev_{10}	$[0, \infty]$	$R_1 = c_2$
ev_{10}	ev_{11}	$[0, \infty]$	-
ev_{11}	ev_5	$[0, \infty]$	$H_1 = c_1$
ev_5	ev_6	$[2, 4]$	$H_1 = c_1$
ev_6	ev_{12}	$[0, \infty]$	$H_1 = c_1$
ev_{11}	ev_7	$[0, \infty]$	$H_1 = c_2$
ev_7	ev_8	$[2, 4]$	$H_1 = c_2$
ev_8	ev_{12}	$[0, \infty]$	$H_1 = c_2$
ev_9	ev_{12}	$[0, 10]$	-

Table B.2: Temporal constraints for the simple example problem.

$H_1 = c_1$	0.1
$H_2 = c_2$	0.9

Table B.3: The conditional probability table (CPT) for our simple influence diagram.

B.2.3 Resultant BDD

After compiling our WMC variables using our Bayesian Inference subsolver, we are left with the Conditional Probability Table in B.4 (as shown in Chapter 5).

H_1	$\Pr(H_1)$	WMC
c_1	0.1	θ_1
c_2	0.9	θ_2

Table B.4: The added WMC vars

yielding the Reduced, Ordered Binary Decision Diagram in Figure B-2.

B.3 Converting Between Modeling Languages

To implement this thesis' work, we found it necessary to translate between two representations of a set of variables \mathcal{V} and constraints. While the implementation details of these modeling changes are not relevant to the theoretical foundation of this thesis,

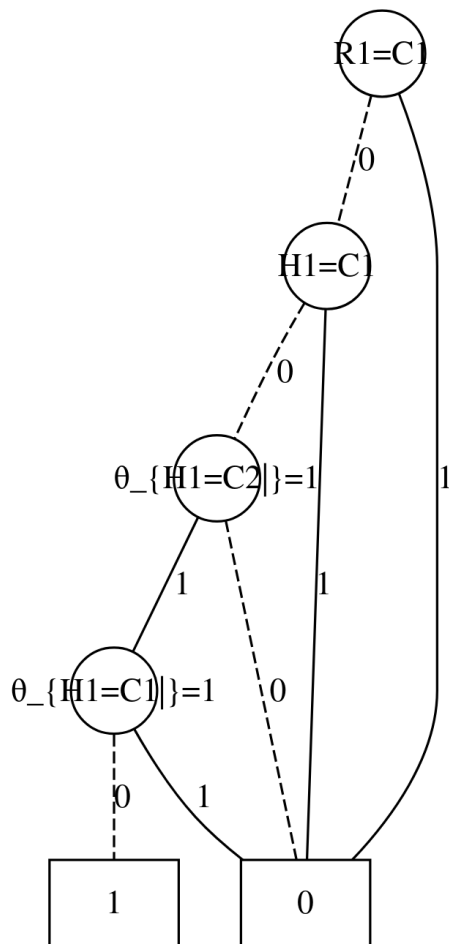


Figure B-2: The final resultant BDD for our example problem.

we would be remiss to not mention the challenges and data structures we used to implement the research.

Here, we'll describe in brief the procedure we use to translate constraints from one modeling language to the other, noting some key differences between the way the two choose to express things, and some of the challenges we encountered during the implementation.

B.3.1 Semantic Differences

Before diving into the technical details of different modeling techniques for CSCSPs and their constituent components, we shall investigate the *semantic differences* of the two modeling languages. Most importantly, the way Odo and RIKER concep-

tualize *conditional constraints* are fundamentally different. In Odo, if a variable is not activated, then all constraints containing this variable are vacuously satisfied. In RIKER, however, constraints must always be satisfied regardless of the activation of the variables in its scope. In practice, this means an assignment to a variable encoded in the CSCSP means that the variable is active.

Consider the constraint from our example problem in Section 4.3.2, $(H_1 = c_1) \Rightarrow ((R_1 = c_1) \wedge (R_2 = c_2))$. In the RIKER modeling language, this constraint can be translated into plain English as “ H_1 is active and assigned c_1 if R_1 is active and assigned c_1 and R_2 is active and assigned c_2 .” In the Odo modeling language, however, this constraint can be translated into plain English as “ H_1 is assigned c_1 if R_1 is assigned c_1 and R_2 is assigned c_2 or any of the variables (H_1 , R_1 , or R_2) is inactive.” The Odo modeling language reflects the traditional conditional CSP approach as outlined in [13]. While the RIKER constraint expression allows us to easily encode disjunctive activation, we choose to enforce the Odo semantic philosophy on RIKER’s constraint expression to bring the semantics in line with those accepted by the constraint modeling community for our CSCSP solver.

To appropriately encode an Odo constraint in Riker, we preface every constraint that contains conditional variables with a “guard” consisting of a conjunction of our induced activity variables. Returning to our simple example, if our Odo constraint c is defined as $(H_1 = c_1) \Rightarrow ((R_1 = c_1) \wedge (R_2 = c_2))$ and R_1 is a conditional variable, then our RIKER constraint c' becomes $\text{active}_{H_1} \Rightarrow ((H_1 = c_1) \Rightarrow ((R_1 = c_1) \wedge (R_2 = c_2)))$.

B.3.2 Conversion into RIKER’s modeling language

In order to convert a CSCSP represented as an Odo CSCSP to a Pike SCSP, it is necessary to convert all of the constituent pieces. This corresponds to Algorithm 10, which takes in a CSCSP represented in the Odo modeling language, compiles out the conditional variables, and represents the resulting SCSP in the RIKER modeling language. In turn, Algorithm 10 relies on Algorithm 11 to create a new state space, set of constraints, influence diagram, risk bound, and variable order expressed in the Riker modeling language. In this way, we can represent a Weighted CSCSP in a

standard modeling language, hand it off to a CSCSP solver, and translate it into a format understood by the RIKER policy generator.

Algorithm 10 CSCSP-CONVERSION

Input: A CSCSP represented in the Odo modeling language

Output: An SCSP represented in the Riker modeling language

- 1: Convert constraints using CONSTRAINT-CONVERSION, compiling out conditional variables
 - 2: Convert the Bayesian network to an influence diagram using INFLUENCE-DIAGRAM-CONVERSION
 - 3: **for** $v_{\text{active}} \in \mathcal{V}$ **do**
 - 4: Add the constraint $(\text{active}_v = \top) \Leftrightarrow \bigwedge_{d_i \neq o} v = d_i$
 - 5: **end for**
 - 6: Convert the chance constraint
 - 7: Use the resulting state space \mathcal{V}' , set of constraints \mathcal{C}' , Δ' , Pr' and inferred variable order $\mathcal{V}_{\text{order}}$ to define an SCSP
-

Algorithm 11 converts constraints from the Odo modeling language into the RIKER modeling language. Our CONSTRAINT-CONVERSION algorithm leverages the recursive constraint structure to maintain a list of variables in a constraint while building up equivalent constraints in the desired modeling language. Taking an example constraint $((H_1 = c_1) \Rightarrow ((R_1 = c_1) \wedge (R_2 = c_1)))$, we can further describe Algorithm 11's helper procedure CONVERSION-HELPER graphically to build up our intuition of the conversion libraries. In short, CONVERSION-HELPER builds up constraints using a modified depth-first search approach, treating each sub-constraint as a sub-tree of the search graph. Equality constraints correspond to the leaf nodes of our search tree; when a leaf node is encountered, the corresponding decision variable assignment is then "passed up" the search tree for integration into the Odo-modeled constraint.

In Figure B-3, **green** corresponds to the constraint conversion. Correctly converted constraints expressed in the RIKER modeling language are denoted with a ' marker. **Blue** denotes the addition of variables into a state space. **Red** corresponds with the (labeled) algorithmic steps by which constraints are recursively converted and generated. The black boxes represent a call to the CONVERSION-HELPER procedure with the given input. Stepping through the graphical representation step-by-step, we take our input constraint $((H_1 = c_1) \Rightarrow ((R_1 = c_1) \wedge (R_2 = c_1)))$, represented in Odo,

Algorithm 11 CONSTRAINT-CONVERSION

Input: c , an Odo constraint, and \mathcal{V}' , an optional RIKER state space

Output: c' , a Pike constraint representing the Odo constraint

```
1: for  $c \in \mathcal{C}$  do
2:    $\mathcal{C}' \leftarrow \text{CONVERSION-HELPER}(c)$ 
3:   if  $c$  contains conditional variables then
4:     Add the proposition  $\text{active}_{v_i}$  for every conditional variable  $v_i \in V(c)$ 
5:     Modify  $c$  into a constraint guarded by  $\text{active}_{v_i} = \top$ 
6:     Add the constraint that ensures if a variable's guard is active, it is assigned
    to a domain value that is not  $\circ$ 
7:   end if
8: end for
9: procedure CONVERSION-HELPER (CH)( $c$ )
10:  if  $c$  is an equality constraint then ▷ Base case.
11:    Convert variable RIKER equivalent
12:    Add variable to RIKER state space.
13:     $c' \leftarrow$  the equivalent Pike assignment constraint
14:  else if  $c$  is an implication constraint then
15:     $c' \leftarrow \{\text{CH}(\text{implicant}) \Rightarrow \text{CH}(\text{consequent})\}$ 
16:  else if  $c$  is an equivalence constraint then
17:     $c' \leftarrow \{\text{CH}(\text{lhs}) \Leftrightarrow \text{CH}(\text{rhs})\}$ 
18:  else if  $c$  is an conjunction constraint then
19:     $c' \leftarrow \{\bigwedge \text{CH}(\text{conjuncts}(c))\}$ 
20:  else if  $c$  is a disjunction constraint then
21:     $c' \leftarrow \{\bigvee \text{CH}(\text{disjuncts}(c))\}$ 
22:  else if  $c$  is a disjunction constraint then
23:     $c' \leftarrow \neg \text{CH}(c)$ 
24:  end if
25: end procedure
```

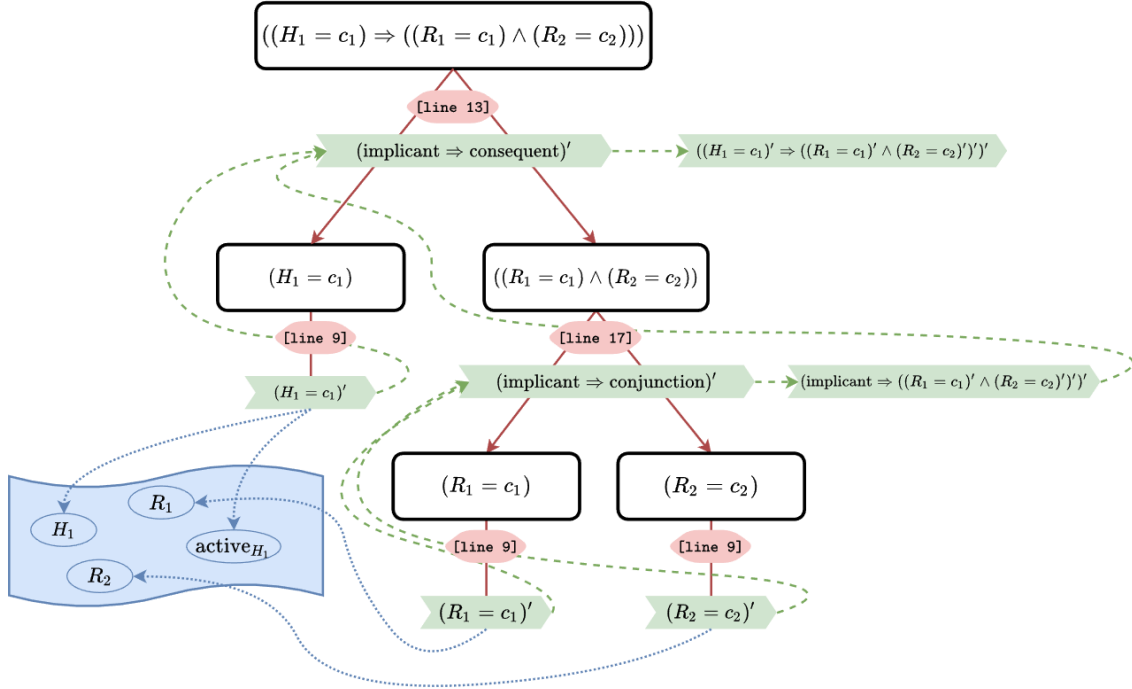


Figure B-3: A visual depiction of the CONSTRAINT-HELPER procedure.

and feed it into the CONVERSION-HELPER procedure. We immediately recognize this is an implication constraint, triggering [line 13](#) of our algorithm, which converts the implication constraint into the appropriate RIKER constraint and calls CONVERSION-HELPER on the contents of the *implicant* and contents of the *consequent*.

Arriving at the second level of the tree, we notice that $(H_1 = c_1)$ is an equality constraint that contains a conditional variable. This is our base case represented in [line 9](#) of Algorithm 11. First, we check to see if a variable with the same name as the one represented in the constraint already exists in the state space of variables. If it does, CONVERSION-HELPER uses the pre-existing variable in the constraint. If a variable with the same name does not already exist, CONVERSION-HELPER will convert the variable to its equivalent RIKER representation using Algorithm 12, [add it to the state space](#), and use this variable in the constraint expression. This check avoids the unnecessary representation of identical variables in the state space and is relatively efficient due to the internal hash table structure of a given state space. Next, CONVERSION-HELPER creates an assignment to the given decision variable and [passes \$\(H_1 = c_1\)'\$ back up](#) to its parent caller. Finally, because the variable is

conditional, we define an auxiliary variable active_{H_1} which takes the value of \top when H_1 is active and \perp otherwise and extend the domain of variable H_1' to be $\{c_1, c_2, \circ\}$. This auxiliary activity variable is also [added into our generated state space](#). These two modifications “compile out” the conditional constraint as outlined in Section 4.3.3.

Now we’ve encountered a leaf node (decision variable assignment), we move in a depth-first fashion over to the right side of our conversion tree to examine the consequent $((R_1 = c_1) \wedge (R_2 = c_2))$. We recognize that this is a conjunction, triggering [line 17](#) of Algorithm 11. Unlike an implication constraint, a conjunction can be formed of a theoretically unlimited amount of conjuncts. To solve this problem, we map the \wedge operator onto a list returned by the CONSTRAINT-HELPER procedure (this procedure corresponds to [line 18](#) of Algorithm 11).

Moving down to our third level of recursion, we notice that our first conjunct is just an equality constraint. Following the steps outlined in [lines 9-12](#) of Algorithm 11, we check to see if this decision variable is already in the [state space, create it if not](#), and [pass the resulting decision variable assignment \$\(R_1 = c_1\)'\$](#) up to the second level of the tree. We perform the same procedure on the $R_2 = c_2$ node, though R_2 is not conditional so we do not need to add an auxiliary activity variable. Now that there are no more conjuncts to explore, we apply our \wedge operator to the list of generated constraints to form $((R_1 = c_1' \wedge (R_2 = c_2)'))'$. This RIKER-modeled constraint is then [passed up to the first level of the search tree](#) and combined with the left side of our search tree to form the converted constraint $((H_1 = c_1)' \Rightarrow ((R_1 = c_1)' \wedge (R_2 = c_2)'))'$.

Now that we have created an equivalent constraint in RIKER using our CONVERSION-HELPER procedure, it is time to modify the semantics as outlined in Section B.3.1 by encoding a guard so that constraints *may* be vacuously satisfied if the variables that appear in the constraint are not encoded. This is done by performing a search over the generated state space to find all of the activity variables and encoding a guard. Our resultant constraint (and state space) is now represented as $\text{active}_{H_1} \Rightarrow ((H_1 = c_1)' \Rightarrow ((R_1 = c_1)' \wedge (R_2 = c_2)'))'$.

Algorithm 12 deals with two problems: first, it translates all Odo variables in to RIKER variables. In parallel, it compiles out conditional variables by adding in an

Algorithm 12 VARIABLE-CONVERSION

Input: v , a variable represented in the Odo modeling language

Output: v' , a variable represented in the RIKER modeling language

- 1: $\mathcal{V} \leftarrow v'$, where v' is the translation from Odo to RIKER
 - 2: **if** v is conditional **then**
 - 3: $D(v') \leftarrow \circ$ ▷ Adds “inactive” value to variable domain
 - 4: $\mathcal{V} \leftarrow \text{active}_{v'}$ ▷ Adds active proposition to the state space
 - 5: **end if**
-

“inactive” value to conditional variable’s domains and the auxiliary active proposition to the state space. By explicitly representing a variable’s activation status in our CSP formulation, we are able to use canonical BDD algorithms to compile out our constraints. Recall that our BDD structure allows us to reason over a huge number of policies to make controllable decision variables; because the BDD does not permit conditional variables, we must compile away the conditional nature of the constraints (as outlined in Section 4.3.3).

Bibliography

- [1] Jacob Broida. *Active Policy Querying for Dynamic Human-Robot Collaboration Tasks*. PhD thesis, Massachusetts Institute of Technology, 2021.
- [2] Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
- [3] Andre Cire and Willem-Jan van Hoeve. Multivalued Decision Diagrams for Sequencing Problems. *Operations Research*, 61, December 2013.
- [4] Patrick R Conrad and Brian C Williams. Drake: An Efficient Executive for Temporal Plans with Choice. *Journal of Artificial Intelligence Research*, 42:53, December 2011.
- [5] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence*, 49(1):61–95, May 1991.
- [6] Cheng Fang. *Efficient algorithms and representations for chance-constrained mixed constraint programming*. Thesis, Massachusetts Institute of Technology, September 2021. Accepted: 2022-02-07T15:12:01Z.
- [7] Esther Gelle and Mihaela Sabin. Solving methods for conditional constraint satisfaction. In *In IJCAI-2003*, pages 7–12, 2003.
- [8] Ronald A. Howard and James E. Matheson. Influence Diagrams. *Decision Analysis*, 2(3):127–143, September 2005. Publisher: INFORMS.
- [9] Finn V. Jensen and Thomas D. Nielsen. *Bayesian Networks and Decision Graphs*. Information Science and Statistics. Springer, New York, NY, 2007.
- [10] A. B. Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, November 1962.
- [11] Steven James Levine. *Risk-bounded coordination of human-robot teams through concurrent intent recognition and adaptation*. Thesis, Massachusetts Institute of Technology, 2019. Accepted: 2019-07-15T20:31:07Z.
- [12] D. Michie. CHAPTER 8 - GAME-PLAYING AND GAME-LEARNING AUTOMATA. In L. Fox, editor, *Advances in Programming and Non-Numerical Computation*, pages 183–200. Pergamon, January 1966.

- [13] Sanjay Mittal and Brian Falkenhainer. Dynamic Constraint Satisfaction Problems, 1990.
- [14] Paul Morris, Nicola Muscettola, and Thierry Vidal. Dynamic control of plans with temporal uncertainty. In *Proceedings of the 17th international joint conference on Artificial intelligence - Volume 1*, IJCAI'01, pages 494–499, San Francisco, CA, USA, August 2001. Morgan Kaufmann Publishers Inc.
- [15] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a Standard CP Modelling Language. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, volume 4741, pages 529–543. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. Series Title: Lecture Notes in Computer Science.
- [16] Ioannis Tsamardinos. A Probabilistic Approach to Robust Execution of Temporal Plans with Uncertainty. In *Proceedings of the Second Hellenic Conference on AI: Methods and Applications of Artificial Intelligence*, SETN '02, pages 97–108, Berlin, Heidelberg, April 2002. Springer-Verlag.
- [17] Toby Walsh. Stochastic constraint programming. In *Proceedings of the 15th European Conference on Artificial Intelligence*, ECAI'02, pages 111–115, NLD, July 2002. IOS Press.
- [18] Brian C. Williams and Robert J. Ragno. Conflict-directed A* and its role in model-based embedded systems. *Discrete Applied Mathematics*, 155(12):1562–1595, June 2007.