Lifelong Verification of Model-Based Programs

by

Paul Harrison Elliott

Submitted to the Department of Aeronautical and Astronautical Engineering
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 2008

Author . . . . . .
Department of Aeronautical and Astronautical Engineering
August 21, 2008

Certified by . . . . . .
Brian C. Williams
Thesis Chair, Professor in Aeronautical and Astronautical Engineering
Thesis Supervisor

Certified by . . . . . .
Howard Shrobe
Principal Research Scientist in Electrical Engineering and Computer Science
Thesis Supervisor

Certified by . . . . . .
David Karger
Professor in Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . . . .
Prof. David L. Darmofal
Associate Department Head
Chair, Committee on Graduate Students

Lifelong Verification of Model-Based Programs

by

Paul Harrison Elliott

Submitted to the Department of Aeronautical and Astronautical Engineering
on August 21, 2008, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

This thesis addresses a problem that arises in model-based autonomy. In model-based autonomy, a probabilistic plant model is used to elevate the mission goals from the level of explicitly actuating the system and evaluating the action based on sensor data to that of specifying the desired state plan. A controller uses the model to support the elevated goals. This work focuses on closing the loop around both the execution of the state plan and the controller, where prior work has only focused on each part separately. This algorithm provides a novel plan monitoring capability and thus predicts the probability that a plan will succeed in the future, using the plant model, in order to determine how the plan will likely evolve. This algorithm is able to incorporate any observations and actuations available from the execution of the plan.

This thesis uses a sampling approach to solve plan monitoring problem, sampling both possible plan executions and the corresponding plant evolutions that could have occurred given the plan execution. This thesis provides three primary novel contributions in its approach. The first is the novel capability of closing the loop of the plan's execution in conjunction with a probabilistic model of the system being controlled. Using this plan monitoring capability, the plan's success can be predicted prior to execution, monitored during execution, and evaluated after execution.

Second, this thesis presents a novel solver that generates and samples the probabilities needed to provide the plan monitoring capability. This solver encodes the solutions of the problem in a decomposable negation normal form (DNNF) representation and includes a DNNF sampling algorithm as well as a algorithm for extracting the k-best solutions from the DNNF.

Finally, this thesis shows how to use this solver to compute the belief state update equations for the probabilistic plant model, called a *probabilistic concurrent constraint automata* (PCCA) model. These PCCA update equations represent a novel contribution of the semantics of the models with respect to state estimation. This is the first approach that allows for non-uniform observation probabilities.

Thesis Supervisor: Brian C. Williams
Title: Thesis Chair, Professor in Aeronautical and Astronautical Engineering

Thesis Supervisor: Howard Shrobe

Title: Principal Research Scientist in Electrical Engineering and Computer Science

Thesis Supervisor: David Karger
Title: Professor in Electrical Engineering and Computer Science

Acknowledgments

I thank my wife Joelle Brichard for her patience.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

In embedded devices that rely on physical components, successful execution can rarely be guaranteed a priori. Instead, success of a program execution depends on the correct functioning of the physical components. The relative health of the components change during the lifetime of the system, hence effecting the correct function of the system over time. Hence, to achieve correctness it is insufficient to simply verify a system at design time. A correct design may quickly degrade to the point where correct function is lost. Instead this thesis proposes a *lifelong* approach to validation, in which software systems continuously verify their correct function against specifications, by estimating the health of the system components, and by verifying functions online by estimating the likelihood of successful execution against these health estimates.

We develop this lifelong verification approach in the context of model-based programming [56]; the intent of a task is separated from the method of accomplishing the task. In traditional practice one specifies small programs or action sequences to accomplish a task. These action sequences specify device actuations and the expected sensor data, possibly with temporal constraints. With model-based programming, the task instead specifies the desired sequence(s) of states, including temporal constraints. A model-based controller then uses a probabilistic model of the system dynamics to map sensor data into states and to map the desired states into the actuation required. Fig. 1-1 shows a typical architecture of an executive for a model-based program.

The separation of task intent from the method of accomplishing the task has the benefit of

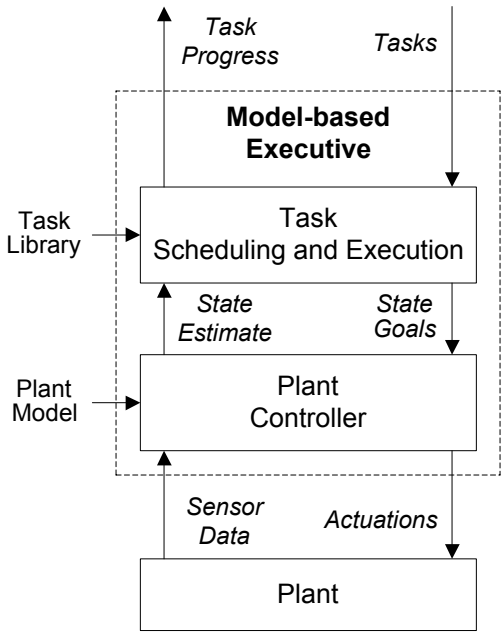

Figure 1-1: The architecture of a model-based executive. The scheduler executes tasks by requesting the controller put the plant into a particular sequence of states that accomplish the task. The scheduler uses the state updates to decide the progress of the task. The controller actuates the plant to move the plant's current state to a state that achieves the immediate goals. The controller assesses the current state by examining the sensor data.

isolating changes to the mission from changes to the device specification. This separation also has the benefit of allowing tasks to ignore minor failures as they can be handled automatically by the model-based controller.

A model-based executive can only handle failures automatically if it is able to both detect failures and adjust for them when possible, this is an instance of closed-loop control. At the level of individual steps in the task, which specify a particular goal region in state-space, a closed-loop executive senses the current state, including failure states, and takes appropriate corrective action to repair the failure, such as power cycling a stalled transmitter. This corrective action moves the current state to a state in the current goal region of the task. The executive also detects if the step cannot be accomplished due to failure. At the level of the task, our closed-loop executive senses the current progress on the task and predicts the future success of the task. The executive can abandon or re-plan tasks that are doomed to fail before the failure occurs.

Currently model-based executives close-the-loop only at the level of each goal in a task. There is a pressing need for a closed-loop executive that monitors at the level of the task. In order to close the loop at the level of the task, and thus be able to verify future success of the task, an executive needs to have three capabilities: (1) monitor the current task progress, (2) predict future task success, and (3) reason about system states that are indirectly observed through sensors. The first capability allows the executive to determine how much of the task has been completed and what needs to be done next. The second capability allows the executive to determine when it is no longer possible to finish the task, so it does not blindly continue. The last capability allows the executive to deal with real-world systems, where state is rarely certain and steps do not complete at a definite time.

To perform closed-loop task monitoring, an executive needs four pieces of information. First is a specification of the task with temporal bounds. Second is a specification of how observations probabilistically map to states. Third is a specification of how each step probabilistically maps into actions, and finally is a time-stamped list of the observations of the task. For a model-based executive [56], the first is specified by a control program, the second and third by employing a plant model, and the last is supplied to the executive as input from the runtime system.

This thesis investigates the problem of lifelong, online verification within the context of task

execution by a model-based executive.

1.1 Motivation

This thesis focuses on embedded systems where human intervention is difficult or impossible, as well as systems where only limited intervention is possible. Examples of these types of systems include satellites and exploratory robots in remote locations such as under the sea, the arctic circles, and in outer space. When intervention is limited, more robust autonomous behavior is required.

It is generally accepted that closed-loop behavior is a desirable trait of all autonomous executives. Closed-loop behavior ensures robustness to disturbances, while the executive works on accomplishing the mission. Executives that do not have this capability are prone to blindly attempt to complete tasks that it cannot accomplish, due to failures that will cause a future part of the task to fail. This can both cause the executive to take on tasks that it cannot accomplish and to continue doing tasks that will fail with certainty in the future. For example, consider a small autonomous submarine with a sensor for mapping the ocean floor. Assume the executive determines that it has less power left than expected, because some step ended up using more power than expected. An open-loop executive is unable to notice that it is unable to complete the task until the system safeguards kick in. A closed-loop executive that verifies future actions is able to identify the problem sooner, and can replan and take on a new task that has a smaller scope to compensate for the problem. This leads to the accomplishment of more of the mission in the face of unexpected events.

Another type of problem that arises in open-loop executives occurs when the controller is unable to achieve the requested task in the time window allotted. This problem arises when the controller attempts to react to unexpected failures. The controllers we are interested in are not aware of the time constraints of the task, and so are unable to themselves indicate if a task cannot be accomplished in the desired window. Flexible task descriptions leave the controller opportunities to recover from some failures successfully, but an appropriate combinations of errors may force the system under control device to be in a state for which the executive can not recover in time. For example, if the controller is managing some machines and one of them stops working, the controller may still be able to fulfill the orders, but not in sufficient time. The controller itself, unaware of the time, will

continue trying to fulfill the order until it runs out of time. The closed-loop executive must recognize that the controller cannot fulfill the task within its time bounds.

In order to close the loop at the level of the task, we need an improved algorithm for mapping sensor data into states, an estimator; the improved algorithm has fixed time and memory bounds as well as improved estimation accuracy. This improvement also benefits the embedded applications of the controller itself. Embedded devices frequently have limited processing and memory resources and prefer to allocate time and memory in a fixed amount to each process a priori. The improvements to the estimator's bounds ensure both that the estimator is always able to do its job and that excessive amounts of resources are not wasted the rest of the time.

Improving the accuracy of the estimator improves the stability of closed-loop control of the system. Improved accuracy also improves the estimates of the task success generated by the algorithm proposed by this thesis. For monitoring roles, improved accuracy reduces the number of false positives and false negatives in terms of identifying failures, reducing the human work load required by the monitor.

1.2 Problem Statement

This thesis investigates the problem of monitoring the progress of a task given a probabilistic model of the underlying plant's behavior. Monitoring includes computing, at each time step, the probability of future task success, current success, and when failure occurs, the most likely failure modes. Solving this problem provides a capability that can notify the task issuer when the task is unlikely to finish, as well as how likely it is that a task was accomplished. The solution must take into account any available observations and evaluates the probability that the plan will succeed or has succeeded.

1.3 Challenges

This problem is computationally difficult due to the combinatorial complexity of the problem. The plant model has a large state space of uncertain states to be estimated. This large state space has a correspondingly large number of state trajectories to be predicted into the future. The state space is

exponential in the number of model variables and the number of state trajectories is exponential in the distance predicted into the future.

Predicting the behavior of the execution into the future is computationally intensive. The plausibility of each state trajectory is determined by the control actions taken by the controller, which are in turn determined by what state the executive believes the plant is in. Since the executive does not know the state exactly, we must ensure that, when predicting the future, our simulated executive has the same ambiguity as the executive would in reality. Simulating this miss-information adds to the computational complexity.

Our task specification also makes the problem difficult as the task specifies only regions in state-space. Since it only specifies a partial state, it is generally impossible to solve the problem by breaking it into multiple pieces and solving them independently. If we could break the problem into independent pieces, the length of each piece would be substantially shorter, reducing the number of potential trajectories that need be considered.

1.4 Approach and Innovations

Prior work has focused on the controller and executive separately. On the controller side, the controller is able to diagnose the correct accomplishment of each goal event in the plan. The controller uses a feedback loop between its diagnosis and reconfiguration parts to move the system into the goal state, which makes the controller robust in the face of failures.

On the executive side in prior work, the executive actively adjusts the schedule of events (steps) based on the completion time of each activity. So long as the activities complete within the requested time frame, the task will succeed. Planners that generate tasks for the executive have begun including predictions as to when each step will complete as a mechanism for predicting task failure. This probability distribution can be based on manual specification or based on some form of underlying model of the activity, such as a path planner for motion-based activities. The key to these approaches is that distributions are fully determined by a start and stop event, and the distribution specifies the time of the stop event. For example, for a navigation task, the path planner can assume the robot is in the location specified by the end of the previous move activity. For our problem, the

probability an activity of a task will succeed depends on an arbitrary subset of the prior events and concurrent steps, along with the amount of time the controller is expected to spend recovering from intermittent failures.

The coupled system must take into account the efficacy of the executive and controller. Both are attempting to manipulate the hidden state of the plant. The state estimates are not exact and thus the efficacy of the decisions of the executive and controller directly impact the probability that they will be able to successfully complete the task.

We seek to solve this problem of predicting a task's probability of success by using the model-based controller as a simulator for the plant as well as part of the executive, as shown in Fig. 1-2. Through simulation, the executive is able to predict what might occur in the future, based on the actions taken by the executive and controller in response to the simulated future. Using the simulated futures, the executive can compare each possible future of the plant against the requirements of the task. Due to the number of possible futures, our approach reduces the amount of futures that need to be considered by only approximating the probability of success through a sampling of possible futures.

Using the controller in conjunction with the executive, the task monitoring capability of this thesis is able to predict task completion at any point during its execution. Before the task starts, the task monitor is able to predict the likelihood of the task succeeding, when executed. During execution, the task monitor is able to use sensor and actuator information to adjust the probability of task success. Finally, after completion of task execution, the task monitor is able to use the complete sequence of observations to predict the likelihood that the task succeeded.

1.4.1 Innovations

This thesis contributes a novel capability of approximating the probability that a task will successfully execute given a probabilistic model of the plant. This thesis contributes four new innovations over prior work in order to provide this capability.

1. This thesis provides a novel sampling algorithm to predict possible plant state evolutions, which are used to decide the likelihood of task success.

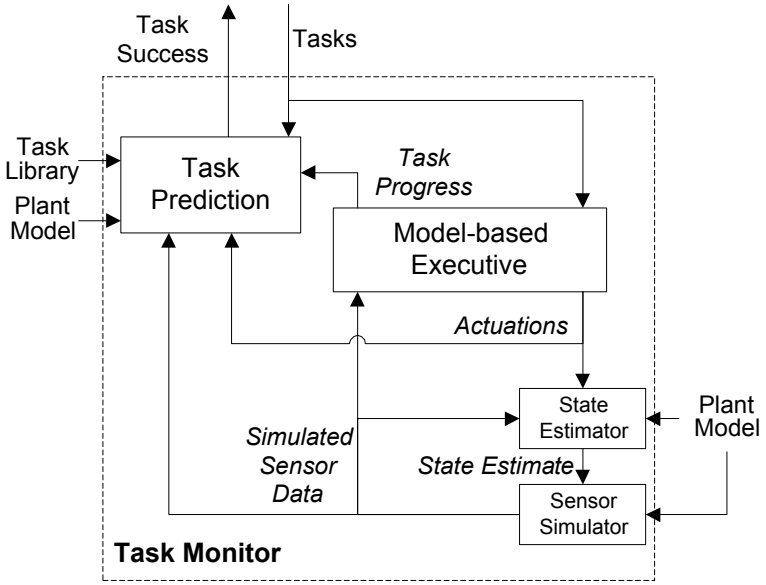

Figure 1-2: This architecture of the task monitoring capability is used to evaluate the probability that a task will succeed when executed. The architecture contains an executive to simulate future trajectories based on possible observation sequences. The task monitor generates plausible observation and command sequences that it then uses to sample possible plant trajectories. A trajectory is then compared against the task's required events to determine if the trajectory accomplishes the task.

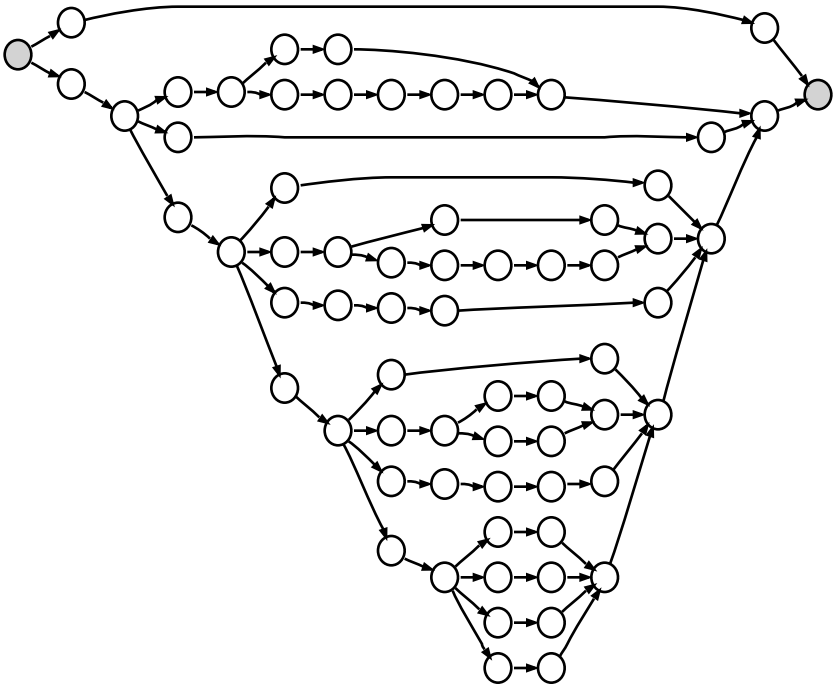

Figure 1-3: A QSP can be visualized as a network in which circles represent events and arrows represent temporal constraints. This simple QSP has a number of parallel sub-tasks that each consist of a few activities.

2. This thesis derives an exact set of estimation equations for probabilistic concurrent constraint automata (PCCA), the plant model used in this thesis.

3. This thesis contributes an efficient algorithm for compiling and computing the estimation and sampling equations that arise in this thesis based on existing decomposition techniques. These equations involve maximizations over sums of products.

4. Approximate estimation is based on a novel algorithm for enumerating the k-best solutions of the equations.

1.4.2 Task Specification

In this thesis, we perform online verification on the deterministic subset of the Reactive Model-based Programming Language (RMPL) [32]. Specifically, the RMPL programs do not include the

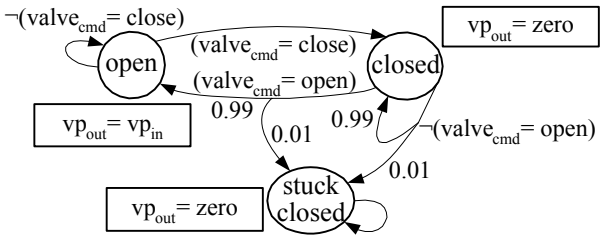

Figure 1-4: This is a simple example of a PCCA model used to describe the state evolution of the plant under control. This model represents a valve that is either open, closed, or stuck closed. The valve can be commanded to open or close and has a possible uncontrollable failure of becoming stuck closed, with a probability of 0.01, when the valve is closed. Each mode of the valve has an associated constraint, for instance when the valve is open the input and output pressures are equal.

conditional choice operator, which allows one to specify alternative ways of accomplishing the program. We can compile a RMPL task into a Qualitative State Plan (QSP). A QSP is a temporally flexible plan specification, which is flexible through the specification of temporal constraints between plan events. Each plan event requires the plant to be in a specific region of state-space at the time of the event. We can visualize a QSP as a Temporal Plan Network (TPN) as shown in Fig. 1-3.

A QSP representation allows for the specification of tasks with some uncertainty as to exactly when certain states need to occur; a QSP is a partial specification of the state trajectories of the plant. The QSP specifies a flexible time window during which the controller must move the plant into a state that is within the requested region of state-space. Scheduling a QSP involves choosing points at which the temporally-flexible events are believed to have occurred, and then issuing appropriate subsequent state-space objectives to the controller.

1.4.3 Plant Model Specification

We model the plant under control as a probabilistic, discrete event system. For this thesis, we use Probabilistic Concurrent Constraint Automata (PCCA) [57] to model the plant. A PCCA model offers a compact encoding of a Hidden Markov Model (HMM), through concurrency and propositional constraints. Fig. 1-4 shows a simple PCCA model of a valve that can get stuck closed.

1.4.4 Executive

We use the plan reformulation and dispatching algorithms from [52] for our scheduling and execution module. This dispatcher handles the scheduling of temporally flexible events, of which some of the events have uncontrollable duration. A controllable event is one in which the scheduler is free to declare that the event has occurred whenever it wants. For instance, if the event is to wait for four to six minutes, the scheduler may declare it is done waiting anytime in that window. An uncontrollable event is one that happens external to the scheduler, and thus the scheduler must determine when the event has occurred from information from the external source. Since our controller is responsible for making each state-change event happen, most events in our system are uncontrollable. Our scheduler attempts to determine when events occurred based on the probability distribution over the possible states provided by the controller.

1.4.5 Controller

The controller consists of an estimator that determines the current state of the plant and a reconfigurator that issues actuation commands so as to move the system from its estimated state to the desired goal state. The estimator infers the current state by reasoning over a model of the system dynamics, the commands that have been executed, and the current sensory observations.

To enable efficient task monitoring, this thesis investigates an efficient estimation algorithm that is tightly coupled to a task monitoring algorithm. In the pursuit of a more efficient and accurate estimation algorithm, this thesis investigates improvements to the *Best-First Belief State Update* (BFBSU) and MEXEC algorithms. The BFBSU algorithm [42][41] improves upon the accuracy of the *Best-First Trajectory Enumeration* (BFTE) algorithm [56], but keeps the potential exponential complexity of the BFTE algorithm. The MEXEC algorithm [2] improves the complexity of the BFTE algorithm to a polynomial time-bound, but keeps the inaccuracies of the BFTE algorithm. We develop a new algorithm with the time bounds of the MEXEC algorithm and the accuracy of the BFBSU algorithm.

The reconfiguration component used in this thesis is described in [56] and [8].

1.5 Roadmap

This thesis develops efficient algorithms for detecting execution failure of model-based programs, diagnosing the cause of the failure, and predicting future success of progress. We start by presenting related work in Chap. 2. This thesis then develops the equations to solve the task monitoring problem in Chap. 3 followed by the algorithms that compute these equations in Chap. 4. This thesis then presents how to compile and evaluate a class of OCSP problems relevant to this thesis in Chap. 5 and Chap. 6. Chap. 7 presents how to use this OCSP solver to compute PCCA Estimation. Finally, we conclude in Chap. 8 with empirical results and conclusions.

Chapter 2

Related Work

In this chapter we present some work, in the field of model checking, which is related to this thesis. All related work from other fields as well as all background material may be found within the most relevant chapter. Model checking focuses on comparing the description of a system, the model, against the requirements that are supposed to hold true for that system. Model checking has focused on both discrete and continuous time model descriptions, and deterministic, non-deterministic, and Markov model descriptions. The statement of correctness for the system has been specified using both Linear Temporal Logic (LTL) and Computational Tree Logic (CTL).

Recall that for our problem we are interested in a stochastic system under *control* towards meeting the objectives of a control program. This can be approximately related to the model checking community as a stochastic system model, such as a Markov Decision Process, with a correctness specification that identifies the correct states over time in a way related to LTL. Our problem has the addition of an (un-modeled) controller providing inputs to the system in order to make the system match the correctness specification, as opposed to the system itself naturally satisfying the correctness specification. This adds a level of complexity to the problem in terms of correctly capturing the behavior of this black-box controller as it interacts with both other specifications.

2.1 Temporal Reasoning

We focus first on the types of temporal reasoning used by model checkers, which as we stated is a method related to our task specification. For this thesis, we focus on a substantially less complex task specification than either the Linear Temporal Logic (LTL) or Computational Tree Logic (CTL) specification used by model checkers, as we need to be able to reason about what needs to be done next to the system in order to continue to achieve the specification. Our tasks compile to a Simple Temporal Network with Uncertainty (STN-u) [22, 23, 55, 54, 45, 52]. STN-u's specifically do not support any type of existential or universal qualifiers, which eliminates the need to search over a number of alternate events that all accomplish the specification in order to find an appropriate one to execute on the system. STN-u's are discussed further in Section 3.2.

Model checking primarily uses CTL or LTL to describe the expected behavior of the system. For a CTL-based solver that includes fairness, see Clarke [26] who shows how to convert an LTL model checking problem to a CTL-with-fairness model checking problem. Clark's approach requires weak fairness, which specifies that all modeled processes are executed infinitely often over an infinite trace.

The model checking community uses the compiled Binary Decision Diagrams (BDDs) [7] representation in conjunction with symbolic model checkers to make solving some types of model checking problems more tractable, as for instance do McMillan's SMV algorithm [43] and Burch's BDD-based version [33] of the Clarke, Emerson, and Sistla algorithm [11]. LTL model checking is also extended to support finite traces instead of infinite traces by Havelund's efficient algorithm [29].

This thesis uses a substantially less complicated representation of tasks because tasks are less complex. However, it does use a compiled representation for the model, as the model itself is complex. This thesis also exclusively focuses on finite traces, as we are only interested in tasks that have a finite maximum duration.

2.2 Model Checking

Model checking of programs [12] involves using a specification of the desired properties of the system, which for our purpose is the task, and a model of the system for which that property holds, in our case the stochastic model. The model checker verifies that, if you start from some specific state s then any valid evolution of the system will meet the specification. In this thesis, we additionally have a third part:a controller that issues commands to the system based on the current (task) specification.

A substantial amount of model checking research has focused principally on problems where a number of discrete choices are possible in the system, specifically as it relates to software and hardware models. For example the SPIN [31] model checker focuses on checking distributed software systems. Model checking of software has addressed the complexity of testing large problems through several techniques, notably through symbolic approaches that use BDDs, as well as bounded model checking that verifies only for a fixed number of steps, and abstracting the problem to something less complicated, possibly with refinement. The Symbolic Analysis Laboratory (SAL) [21], for example, supports all of these different model checking tools.

Symbolic model checking [43] involves representing the formulas and states in a compact BDD form such that the various properties can be combined and tested on the compact representation rather than by testing each possible state explicitly. Each test of the property is able to test sets of related states. McMillan [43] supports the CTL language. Clarke [26] extends the SMV algorithm of McMillan [43] to support LTL instead of CTL, though LTL generally requires more memory to represent comparable CTL specifications. Our task specification is sufficiently simple that a symbolic representation is unnecessarily complex without providing an appreciable algorithmic improvement.

Bounded Model Checking (BMC) [4] involves encoding LTL model checking problems as propositional formula for a finite horizon k and then using a SAT solver to solve them. BMC solvers often require less memory than symbolic model checkers and are also often better at finding counter examples, while symbolic model checkers are often able to prove that properties are true more efficiently. Cimatti [10] showed how to improve upon the encoding of the BMC problem by introducing a more compact propositional logic representation. Cimatti [9] then showed how

to leverage both BDD-based Symbolic model checkers and BMC solvers in the NuSMV model checker.

McMillan [44] proposed an extension to BMC solvers in order to allow them to test unbounded model checking problems ($k = \infty$). This approach is able to verify positive instances in some cases substantially more efficiently than BDD-based approaches.

We are particular interested in two advances over standard model checking. The first is the extension of model checking to probabilistic models by Vardi [53]. Vardi supports probabilistic model checking by converting the model into a Büchi automata that can then be simulated and decide the model checking problem. Vardi's mapping is limited as it potentially uses exponential space, depending on the specification being converted.

The recent PRISM [37, 36, 39] model checker is developed to analyze probabilistic systems. It supports discrete-time Markov chains, continuous-time Markov chains, and Markov decision processes. It uses probabilistic CTL (PCTL) with fairness constraints to describe correct behavior. Kwiatkowska [38] presents a method of abstracting MDPs for the purpose of improving the performance of model checking. As stated above, this abstraction technique has already been shown to be an effective means to improve algorithmic performance for non-probabilistic model checking problems and is likewise shown effective for probabilistic models. For this thesis, we are using a form of discrete-time Markov chains as our model representation, though the actual Markov terms are hard to compute, so we spend a substantial amount of effort in this thesis towards computing these terms faster. If we had a model for the controller, it is possible that we could encode the sufficiently small problems that this thesis addresses in the PRISM model checker.

The second extension of interest are the advances in runtime model checkers, which specifically focus on analyzing finite traces, a sub-problem of this thesis. Gerth [27] modified the algorithms which translate LTL to Büchi automata into an algorithm that can generate the automata on an as-needed basis. The automata can thus be incrementally evaluated as the model itself is evolving. Giannakopoulou [28] also extended the LTL to Büchi automata algorithms to support checking finite traces taken from actual running programs. This work also allows the verification of programs as they run in order to detect failures early. Both of these advances focus on non-deterministic systems,

rather than on the probabilistic systems on which we are focusing.

2.3 Conclusion

Model checking is a field related to our problem of predicting task success since they share the notion of matching a specification against a model. Model checking uses more complex task specification languages that are by nature more difficult to analyze than the languages used in this thesis, but they also assume a more easily evaluated model specification than is used in this thesis. Our work additionally includes a controller that is actively attempting to make the system match the specification, which is a feature that none of the model checking approaches explicitly support. It is still an open question whether the controller used in this thesis can be modeled for the model checker.

Chapter 3

The Task Monitoring Problem

Recall that the objective of this thesis is to develop a capability for the lifelong verification of a model-based program written in the Reactive Model-based Programming Language (RMPL) [32, 56]. The objective of this chapter is to develop the equations necessary to verify the program over the life of its execution. This thesis provides a lifelong verification capability by providing a capability that computes the probability that the plan will finish successfully at any point during the execution of the plan. A program is correct as long as it is probable that the plan will execute successfully. This capability accounts for all available observations.

As stated in the introduction, we use a restricted form of RMPL to specify our programs; specifically, the compiled RMPL tasks cannot contain choices. They can, however, include parallelism and serialization, along with flexible temporal bounds. An example of such a program is shown in Fig. 3-1. We compile an RMPL program into a representation that is efficient for monitoring, called a Qualitative State Plan (QSP) [32, 56], as discussed in Section 3.2. We then monitor the QSP relative to sensor values to determine program execution success. This chapter specifically develops the equations for QSP monitoring: the probability that the QSP will succeed.

In order to monitor a program, it is necessary to have a model of the system being driven, an initial belief state, and any sensor and actuation data available. Prior to executing a program, when there is no sensor and actuation data available, this thesis is capable of predicting the future success of the program. After executing a program, when all the sensor and actuation data is available, this

```
OpNav() {
  parallel {
    try (((cam == On) && (engine == StandBy)))
    monitor {
      when (((cam == On) && (engine == StandBy))) {
        sequence {
          cam.TakePicture(1);
          cam.TakePicture(2);
          cam.TakePicture(3);
          parallel {
            try ((cam == Off));
            cam.ComputeCorrection();
          }
        }
      };
      until ((OpNavError)) {
        OpNav()
      };
      until ((cam == Error)) {
        OpNavFailed()
      }
    }
  }
}
```

Figure 3-1: An example RMPL program. This program puts the engine in standby mode while making sure the camera is ready to continue. It then takes a series of pictures and finally sets the camera back into the off position. While executing, the program handles exceptions with the camera or engine.

thesis is capable of assessing the probability that the program succeeded. Finally, with partial sensor and actuation data, this thesis predicts the probability that the plan will succeed, given the current point of execution.

We first define the notation of this thesis and then define a QSP. Finally, in this chapter, we present the successful execution of a QSP.

3.1 Notation

RMPL programs operate on variables with finite discrete domains, and assignments to these variables from their domains. We denote a single variable with a capital letter X. We denote the domain of this variable $\mathbb{D}_X$. We denote a value from the domain with a lower-case letter, $x \in \mathbb{D}_X$. An assignment is of the form $X = x$. When the variable is not ambiguous, we just use the value x.

Most of the time, we deal with multiple variables. We denote a vector of variables $\mathbf{X}$. We denote the set of all combinations of values that can be assigned these variables as $\mathbb{D}_{\mathbf{X}}$, where $\mathbb{D}_{\mathbf{X}}$ is the cross-product of $\mathbb{D}_{X_i}$ for every $X_i \in \mathbf{X}$. A value vector $\mathbf{x} \in \mathbb{D}_{\mathbf{X}}$ is assigned to $\mathbf{X}$ by the notation $\mathbf{X} = \mathbf{x}$, where this means that each variable in $\mathbf{X}$ is assigned the corresponding value from $\mathbf{x}$. Again, if the variables are not ambiguous, we denote an assignment $\mathbf{x}$.

We use the notation $X^t = x$ or just x^t to indicate that the assignment occurs at time t. To indicate a range of time, say from 0 to t, we use the notation $x^{0:t}$.

3.2 Qualitative State Plan

In this section we review qualitative state plans and the notation specific to plans that we use in the next two chapters. We start with an RMPL program [32] and us the RMPL compiler [32] to compile RMPL to a Qualitative State Plan (QSP) [32, 56, 30]. The compiler embeds the QSP in a Temporal Plan Network (TPN) [58] for use with the planner Kirk [58, 6, 24]. Kirk generates a temporally flexible schedule, a Simple Temporal Network with Uncertainty (STN-u) [22, 23, 55, 54, 45, 52], from the TPN. We then dispatch this STN-u using the sequencer [52].

We now motivate the use of temporally flexible schedules. Programs are dispatched on systems

that are not in general deterministic. For example, disturbances may increase the time it takes to accomplish a particular step in the program. Uncertainty in the situation may also make it impossible to determine exactly how long a step will take to finish. Temporally flexible schedules allow the executive the flexibility to be sensitive to the actual completion time of each step with out the use of extensive wait steps. The executive adjust the start of future steps according to the actual completion times. Thus, this work focuses on executives that can dispatch temporally flexible programs to improve robustness.

We compiled programs written in RMPL into a QSP using an RMPL Compiler [32]. A QSP specifies a desired evolution of the state of the plant over time. Using the notation from Hofmann [30], a QSP consists of a set of events E, a set of activities A, and a set of temporal constraints on events TC. An event $\mathrm{ev} \in E$ represents a fixed point in time. Since we use a discrete-time plant model, our events are fixed to integer values.

An activity is a tuple $\langle \mathrm{ev}_s, \mathrm{ev}_f, \boldsymbol{\sigma}_{goal}, \mathcal{D} \rangle$.

- ev_s and ev_f are the events that represent the start and finish, respectively, of the activity.

- $\boldsymbol{\sigma}_{goal}$ specifies the region of the state-space that must hold for the activity to finish[1]. The plant model is factored into multiple state variables, and so the plant model's state $\mathbf{m}$ is an assignment to the state variables of the plant. A state-space region is an assignment to a subset of these variables.

- The description $\mathcal{D}$ describes how to achieve $\boldsymbol{\sigma}_{goal}$ by the time of the end event. In the computationally simple case, $\mathcal{D}$ is a sequence of commands to issue to the actuators of the plant. At the other extreme, $\mathcal{D} = \boldsymbol{\sigma}_{goal}$, requiring a planner to discover an appropriate command sequence that moves the state from its current value to a state $\mathbf{m}$ that includes $\boldsymbol{\sigma}_{goal}$. The command generation is done online in this latter case to maximize the ability of the controller to accommodate disturbances.

[1]For this thesis, we focus on the controller [56, 8]. This controller reconfigures the plant's state to match the current state goal. This controller does not support constraining the intermediate steps taken to reconfigure the plant; hence, we only develop support for end-in state constraints. Support for state constraints over the whole episode are trivial to support given the approach of this thesis by simply adding them to the Traj algorithm in Section 4.1.2.

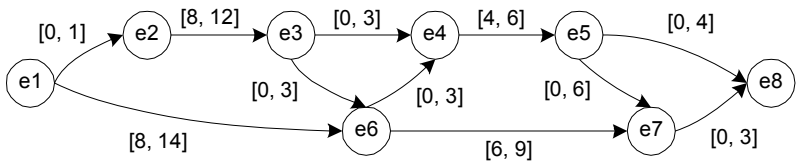

Figure 3-2: This figure shows an example of an STN-u. An STN-u always has a unique start (e1) and end (e8) event and some number of events (circles) and constraints (arcs) in between. Constraints are labeled with the lower (l) and upper (u) time bounds allowed between the two events, denoted [l, u]. The arrow of the arc indicates which event comes first. Events may be labeled with additional information, such as commands that should be issued at the occurrence of the event or values that are expected to be true at the fixing of the event. This STN-u has an execution time of between 14 and 26 steps.

Temporal constraints specify the duration between pairs of events. A temporal constraint is the tuple $\langle \mathrm{ev}_s, \mathrm{ev}_f, l, u \rangle$. ev_s and ev_f are the start and finish events, respectively. l and u specify the lower and upper bounds, respectively, of the duration between the two events such that $l, u \in \{-\infty\} \cup \mathbb{Z} \cup \{\infty\}$. A temporal constraint specifies that $l \leq \mathrm{ev}_f - \mathrm{ev}_s \leq u$.

A consistent schedule for a QSP is an assignment of a fixed time to the events of the QSP such that none of the temporal constraints are violated. A temporally flexible schedule allows the dispatcher some flexibility in when each event can occur, subject to the actual execution still being a consistent schedule. We use Kirk [6, 24] to generate a temporally flexible schedule, a Simple Temporal Network with Uncertainty (STN-u) [55, 54, 45, 52], from the QSP. An STN-u can be visualized as a graph or network by letting the events be nodes and the temporal constraints be arcs, and example of which is shown in Fig. 3-2.

For the purpose of this thesis, we focus on programs for which it is easy to test if the state trajectory of the plant is a member of the state trajectories accepted by the program. The approach of this thesis can be extended to more complex programs by extending the algorithm that tests for membership, Traj, in Section 4.1.2.

3.3 Task Monitoring Problem

In the remaining chapters we will refer to task monitoring as the problem of computing the probability that the qualitative state plan will succeed, or just that the plan will succeed. We define the probability that a plan will succeed in two parts: (1) the probability that the plan will succeed, (2) the probability that a plan succeeded, given a complete execution of the plan. Part (1), the probability that the plan will succeed, reduces to part (2), the probability that the plan succeeded, for a particular observation sequence. Part (1) computes all such observation sequences. Part (2) computes the probability of all trajectories that accomplish the plan.

Our insight is that the QSP describes a set of acceptable state trajectories and thus to determine if the plan will succeed we need to determine how likely it is that the plant follows one of the acceptable trajectories under the active control of the executive.

The QSP specifies multiple acceptable trajectories in three ways: (1) each event specifies a partial state constraint and thus multiple states can satisfy the event, (2) events have temporal flexibility, and thus different trajectories can satisfy the event at different points in time, and (3) in between events, the state is unconstrained.

A **consistent schedule** of an QSP is a temporal assignment to all of the events of the QSP that is consistent with the temporal constraints. A **state trajectory** is a sequence of states through which the system traverses. We denote this trajectory $\mathbf{m}^{0:n}$ and the ith state in the sequence as $\mathbf{m}^i$.

A *state trajectory* is **accepted** with respect to a QSP if the QSP admits the trajectory as a successful execution of the QSP. Specifically, the trajectory is accepted if there is a *consistent schedule* to the QSP such that for each end event fixed at time i, the partial state $\boldsymbol{\sigma}_{goal}$ of the event is a subset of the state $\mathbf{m}^i$. The set of all trajectories accepted by the plan is the union of the set of all accepted trajectories for every consistent schedule. We denote the set of all trajectories accepted by the plan as $\mathrm{Trajs}\,(\mathrm{QSP})$.

Fig. 3-3 depicts an example of an accepted trajectory. In the figure, the trajectory depicted at the bottom aligns with the plan's events at the corresponding vertical dashed lines, and this point of alignment is consistent with the temporal constraints.

If we know the actual state trajectory followed during execution, then we can determine if the

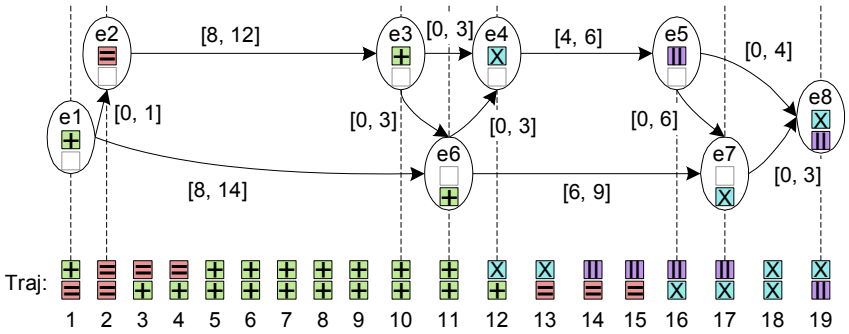

Figure 3-3: This figure depicts a state trajectory (below) with 18 steps that is accepted by the plan (above). The trajectory at the bottom of the figure corresponds to a hypothetical evolution of a system factored into two state variables. Each variable's value is represented as a box, where the possible states are a green $+$, a red $=$, a blue X or a purple $||$. Then plan specifies full and partial state constraints at each event, where a partial constraint only specifies a value for one of the state variables. The unspecified variable is represented as a faint, white box. The trajectory is accepted by the plan because there exists a consistent schedule of the events of the plan such that the events line up with the trajectory, as depicted by the vertical dashed lines.

trajectory accomplishes the plan by determining if it is a member of the set of all trajectories that are accepted by the plan. In general, though, multiple trajectories are possible and we must consider which of them are members of $\mathrm{Trajs}\,(\mathrm{QSP})$.

In order to know which trajectories are likely during execution, it is necessary to know the control inputs. Since the plant is actively controlled by a model-based executive that maps observations to actions based on the plan, this implies simulating realistic observations in order to determine how the executive reacts. The simulation iteratively considers possible observations and the controller's response over the remaining length of the plan. The likely trajectories are those that are both consistent with the simulated observations and the control inputs. This simulation architecture is shown in Fig. 1-2 and repeated here as Fig. 3-4, for simplicity.

We can now more formally define the two parts of the probability that a plan will succeed. Let n be the time at which the final event of the plan is scheduled. For some $t \leq n$, the probability that the plan will succeed is defined as $\mathbf{P}\left(QSP_{succ}|\mathbf{o}^{1:t}, \boldsymbol{\mu}^{1:t}\right)$ and the probability that the plan succeeded is defined as $\mathbf{P}\left(QSP_{succ}|\mathbf{o}^{1:n}, \boldsymbol{\mu}^{1:n}\right)$. Both of these probabilities are also subject to an initial belief state, a plant model, and a plan. We next define how to compute the probability that

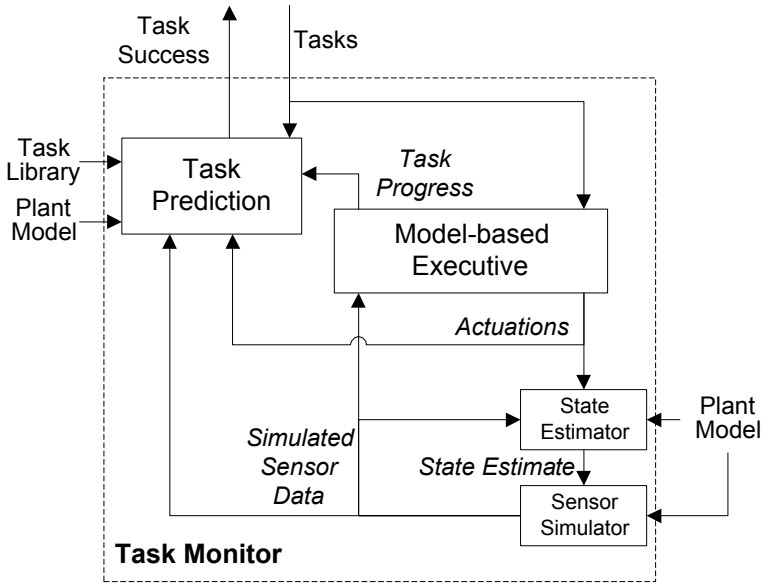

Figure 3-4: This architecture is used to evaluate the probability that a task will succeed. The architecture simulate future behavior in response to possible observation sequences using the actual executive. Given the command sequence generated by the simulation, the Task Prediction module can compare possible trajectories against those accepted by the task.

the plan succeeded:

$$\mathbf{P}\left(QSP_{succ}|\mathbf{o}^{1:n},\boldsymbol{\mu}^{1:n}\right)=\sum_{\mathbf{m}^{0:n}\in\text{Trajs}(QSP)}\mathbf{P}\left(\mathbf{m}^{0:n}|\mathbf{o}^{1:n},\boldsymbol{\mu}^{1:n}\right) \tag{3.1}$$

and then show how to compute the probability that the plan will succeed:

$$\mathbf{P}\left(QSP_{succ}|\mathbf{o}^{1:t},\boldsymbol{\mu}^{1:t}\right)=\sum_{\mathbf{o}^{t+1:n}\in\mathbb{D}_{\mathbf{O}^{t+1:n}}}\mathbf{P}\left(\mathbf{o}^{t+1:n}|\mathbf{o}^{1:t},\boldsymbol{\mu}^{1:t}\right)\mathbf{P}\left(QSP_{succ}|\mathbf{o}^{1:n},\boldsymbol{\mu}^{1:n}\right) \tag{3.2}$$

3.3.1 Probability that the Plan Succeeded

To evaluate the probability that the plan succeeded after execution, $\mathbf{P}\left(QSP_{succ}|\mathbf{o}^{1:n},\boldsymbol{\mu}^{1:n}\right)$, we examine possible trajectories given the observation and control sequence. The idea is to accumulate all the trajectories accepted by the plan and compute the probability that the plant followed one of the accepted trajectories:

$$\mathbf{P}\left(QSP_{succ}|\mathbf{o}^{1:n},\boldsymbol{\mu}^{1:n}\right)=\sum_{\mathbf{m}^{0:n}\in\text{Trajs(QSP)}}\mathbf{P}\left(\mathbf{m}^{0:n}|\mathbf{o}^{1:n},\boldsymbol{\mu}^{1:n}\right). \tag{3.3}$$

We assume we have an estimation algorithm that can compute the necessary probabilities in this chapter, specifically the belief state estimates, the probability of an observation given the state, and the probability of a single step forward. A *Belief State* B^i is a function that maps a state $\mathbf{m}$ to the probability that $\mathbf{m}$ is the system's actual state at time i; $B^i\left(\mathbf{m}^i\right)=\mathbf{P}\left(\mathbf{m}^i|\mathbf{o}^{1:i},\boldsymbol{\mu}^{1:i}\right)$. We use B^i_P to denote the belief state at time i *predicted* from the belief state B^{i-1} given the command at time i but without observation at time i; $B^i_P\left(\mathbf{m}^i\right)=\mathbf{P}\left(\mathbf{m}^i|\mathbf{o}^{1:i-1},\boldsymbol{\mu}^{1:i}\right)$.

The probability of a trajectory $\mathbf{m}^{0:n}$ depends on the belief states $B^{0:n}$ and the sequence of observations and commands from time 0 to n. We compute $\mathbf{P}\left(\mathbf{m}^{0:n}|\mathbf{o}^{1:n},\boldsymbol{\mu}^{1:n}\right)$ using the smoothing equations of the Rauch-Tung-Striebel Smoother (RTSS) [47]. RTSS compute the probability of a state $\mathbf{m}^t$ given all of the evidence up to time $n\geq t$, specifically $\mathbf{o}^{1:n}$ and $\boldsymbol{\mu}^{1:n}$. This probability

$\mathbf{P}\left(\mathbf{m}^t|\mathbf{o}^{1:n}, \boldsymbol{\mu}^{1:n}\right)$ is designated $\mathbf{P}^S\left(\mathbf{m}^t\right)$. RTSS specifies that:

$$\begin{aligned}\mathbf{P}^S\left(\mathbf{m}^t\right) &= \mathbf{P}\left(\mathbf{m}^t|\mathbf{o}^{1:t}, \boldsymbol{\mu}^{1:t}\right) \sum_{\mathbf{m}^{t+1}\in\mathbb{D}_{\mathbf{M}^{t+1}}} \frac{\mathbf{P}\left(\mathbf{m}^{t+1}|\mathbf{m}^t, \boldsymbol{\mu}^{t+1}\right)}{\mathbf{P}\left(\mathbf{m}^{t+1}|\mathbf{o}^{1:t}, \boldsymbol{\mu}^{1:t+1}\right)} \mathbf{P}^S\left(\mathbf{m}^{t+1}\right) \qquad (3.4)\\ &= B^t\left(\mathbf{m}^t\right) \sum_{\mathbf{m}^{t+1}\in\mathbb{D}_{\mathbf{M}^{t+1}}} \frac{\mathbf{P}\left(\mathbf{m}^{t+1}|\mathbf{m}^t, \boldsymbol{\mu}^{t+1}\right)}{B_P^{t+1}\left(\mathbf{m}^{t+1}\right)} \mathbf{P}^S\left(\mathbf{m}^{t+1}\right)\end{aligned}$$

$$\mathbf{P}^S\left(\mathbf{m}^n\right) = \mathbf{P}\left(\mathbf{m}^n|\mathbf{o}^{1:n}, \boldsymbol{\mu}^{1:n}\right) = B^n\left(\mathbf{m}^n\right) \qquad (3.5)$$

All three probabilities of Eq. 3.4 are computed by the estimation algorithm. The first term of both equations correspond to B^t and the denominator of Eq. 3.4 corresponds to B_P^{t+1}. The transition probability $\mathbf{P}\left(\mathbf{m}^{t+1}|\mathbf{m}^t, \boldsymbol{\mu}^t\right)$ is computed as part of the estimator's belief state update calculation. $\mathbf{P}^S\left(\mathbf{m}^n\right)$ is the final probability distribution at the end of the plan. In order to convert these equations into a form for computing the probability of a trajectory we note that a trajectory specifies a specific next state $\mathbf{m}^{t+1}$, so we can eliminate the summation from Eq. 3.4 and compute $\mathbf{P}^S\left(\mathbf{m}^0\right)$. For a trajectory $\mathbf{m}^{0:n}$ and $\mathbf{m}^t, \mathbf{m}^{t+1} \in \mathbf{m}^{0:n}$:

$$\begin{aligned}\mathbf{P}^S_{\mathbf{m}^{0:n}}\left(\mathbf{m}^t\right) &= \mathbf{P}\left(\mathbf{m}^t|\mathbf{o}^{1:t}, \boldsymbol{\mu}^{1:t}\right) \frac{\mathbf{P}\left(\mathbf{m}^{t+1}|\mathbf{m}^t, \boldsymbol{\mu}^{t+1}\right)}{\mathbf{P}\left(\mathbf{m}^{t+1}|\mathbf{o}^{1:t}, \boldsymbol{\mu}^{1:t+1}\right)} \mathbf{P}^S_{\mathbf{m}^{0:n}}\left(\mathbf{m}^{t+1}\right) \qquad (3.6)\\ &= B^t\left(\mathbf{m}^t\right) \frac{\mathbf{P}\left(\mathbf{m}^{t+1}|\mathbf{m}^t, \boldsymbol{\mu}^{t+1}\right)}{B_P^{t+1}\left(\mathbf{m}^{t+1}\right)} \mathbf{P}^S_{\mathbf{m}^{0:n}}\left(\mathbf{m}^{t+1}\right)\end{aligned}$$

$$\mathbf{P}^S_{\mathbf{m}^{0:n}}\left(\mathbf{m}^n\right) = \mathbf{P}\left(\mathbf{m}^n|\mathbf{o}^{1:n}, \boldsymbol{\mu}^{1:n}\right) \qquad (3.7)$$

$$\mathbf{P}\left(\mathbf{m}^{0:n}|\mathbf{o}^{1:n}, \boldsymbol{\mu}^{1:n}\right) = \mathbf{P}^S_{\mathbf{m}^{0:n}}\left(\mathbf{m}^0\right) \qquad (3.8)$$

We substitute Eq. 3.8 into Eq. 3.3 and expand the recursion, yielding our final result, the probability that the plan succeeded:

$$
\begin{aligned}
&\mathbf{P}\left(QSP_{succ}|\mathbf{o}^{1:n}, \boldsymbol{\mu}^{1:n}\right) \\
&= \sum_{\mathbf{m}^{0:n} \in \text{Trajs(QSP)}} \left(\prod_{i=0}^{n-1} \mathbf{P}\left(\mathbf{m}^i|\mathbf{o}^{1:i}, \boldsymbol{\mu}^{1:i}\right) \frac{\mathbf{P}\left(\mathbf{m}^{i+1}|\mathbf{m}^i, \boldsymbol{\mu}^{i+1}\right)}{\mathbf{P}\left(\mathbf{m}^{i+1}|\mathbf{o}^{1:i}, \boldsymbol{\mu}^{1:i+1}\right)} \right) \mathbf{P}\left(\mathbf{m}^n|\mathbf{o}^{1:n}, \boldsymbol{\mu}^{1:n}\right) \qquad (3.9) \\
&= \sum_{\mathbf{m}^{0:n} \in \text{Trajs(QSP)}} \left(\prod_{i=0}^{n-1} B^i\left(\mathbf{m}^i\right) \frac{\mathbf{P}\left(\mathbf{m}^{i+1}|\mathbf{m}^i, \boldsymbol{\mu}^{i+1}\right)}{B_P^{i+1}\left(\mathbf{m}^{i+1}\right)} \right) B^n\left(\mathbf{m}^n\right)
\end{aligned}
$$

3.3.2 Probability that the Plan Will Succeed

For the general task monitoring problem, we are given only B^0 and we want to compute the probability the plan will succeed given some, potentially empty, observation sequence: $\mathbf{P}\big(QSP_{succ}|\, \mathbf{o}^{1:t}, \boldsymbol{\mu}^{1:t}\big)$. To reduce this problem to the problem of the previous section, computing the probability that the plan succeeded, requires predicting future observation and command sequences. Recall that we are using an executive whose future commands depend on the future observations. A realistic observation sequence is necessary to obtain a realistic command sequence.

In order for our observation sequence to be realistic, we need a model for the expected observations. Fortunately, our plant model can provide this probability, and the observation probability distribution at time t only depends on the state at time t, as shown in Sec. 7.3. We compute this probability distribution from our plant model and our predicted belief state at time i: $\mathbf{P}\left(\mathbf{o}^i|B_P^i\right)$. Partial observations can be handled in prediction if the probability of receiving a partial vs. complete observation is provided. For example, we may know that a sensor only provides a measurement every fifth time step. For the purpose of this thesis, we consider only complete observations for prediction, but it is trivial to add in partial observations.

We assume the executive is deterministic in that for a given plan, initial belief state, and observation sequence, it will generate the same command sequence. Since the executive internally generates a belief state for each point in time t and we need this same belief state to compute the realistic observation probability distribution, our simulation uses the belief state estimates generated by the controller to compute the observation distribution. This simulation approach is depicted in

the architecture diagram of Fig. 1-2, repeated above as Fig. 3-4 for simplicity.

For each observation chosen from the probability distribution and the command output by the executive, we compute our next belief state, enabling us to generate another observation probability distribution. We can iteratively branch on possible observations in the observation distribution until we reach a point where the executive indicates the plan is done being dispatched. At this point the problem of computing the probability that the plan will succeed is reduced to the problem of computing the probability that the plan succeeded for this simulated execution. The overall predicted probability is the weighted sum of each simulation's probability that the plan succeeded, where the weight is the probability of each observation used in the simulation. Said another way, the probability the plan will succeed is the summation over all possible observation sequences of the probability of that sequence times the probability that the sequence results in a successful execution of the plan. The commands used in this equation are those generated by a simulation of the controller. In equation form, this probability that the plan will succeed given no observations is:

$$\begin{aligned}\mathbf{P}\left(QSP_{succ}\right) &= \sum_{\mathbf{o}^{1:n}\in\mathbb{D}_{\mathbf{O}^{1:n}}} \prod_{i=1}^{n} \mathbf{P}\left(\mathbf{o}^{i}|B_P^i\right)\cdot\mathbf{P}\left(QSP_{succ}|\mathbf{o}^{1:n},\boldsymbol{\mu}^{1:n}\right) \\ &= \sum_{\mathbf{o}^{1:n}\in\mathbb{D}_{\mathbf{O}^{1:n}}} \prod_{i=1}^{n} \mathbf{P}\left(\mathbf{o}^{i}|B_P^i\right)\cdot \sum_{\mathbf{m}^{0:n}\in\text{Trajs(QSP)}} \mathbf{P}\left(\mathbf{m}^{0:n}|\mathbf{o}^{1:n},\boldsymbol{\mu}^{1:n}\right)\end{aligned} \tag{3.10}$$

To compute the probability that the plan will succeed in the future given some observations and commands up to time t, $0 \leq t \leq n$, we reduce the length of the outer sum and the product of Eq. 3.10. Our final result is:

$$\mathbf{P}\left(QSP_{succ}|\mathbf{o}^{1:t},\boldsymbol{\mu}^{1:t}\right) = \sum_{\mathbf{o}^{t+1:n}\in\mathbb{D}_{\mathbf{O}^{t+1:n}}} \prod_{i=t+1}^{n} \mathbf{P}\left(\mathbf{o}^{i}|B_P^i\right)\cdot\mathbf{P}\left(QSP_{succ}|\mathbf{o}^{1:n},\boldsymbol{\mu}^{1:n}\right). \tag{3.11}$$

Eq. 3.11 reduces to Eq. 3.10 when $t = 0$ and thus there are no observations available. Eq. 3.11 reduces to Eq. 3.3 when $t = n$ and thus there is a complete sequence of observations available.

Combining equations 3.9 and 3.11 yields our overall result:

$$\mathbf{P}\left(QSP_{succ}|\mathbf{o}^{1:t},\boldsymbol{\mu}^{1:t}\right) = \sum_{\mathbf{o}^{t+1:n}\in\mathbb{D}_{\mathbf{O}^{t+1:n}}} \prod_{i=t+1}^{n} \mathbf{P}\left(\mathbf{o}^{i}|B_P^i\right) \cdot \sum_{\mathbf{m}^{0:n}\in\text{Trajs(QSP)}} \left(\prod_{i=0}^{n-1} B^i\left(\mathbf{m}^i\right) \frac{\mathbf{P}\left(\mathbf{m}^{i+1}|\mathbf{m}^i,\boldsymbol{\mu}^{i+1}\right)}{B_P^{i+1}\left(\mathbf{m}^{i+1}\right)}\right) B^n\left(\mathbf{m}^n\right) \tag{3.12}$$

As one can expect from the formulation of these equations, it is in general both intractable to enumerate all observation sequences as well as to enumerate all transition sequences. Thus, this thesis explores in the next chapter an approximate method that uses samples from both sequences to estimate this overall probability.

3.4 Conclusion

This chapter presented the QSP used in this thesis as a set of temporally constrained events and a set of activities between events that specify a state in which the plant must be at the end of the activity.

This chapter also presented a derivation of the probability of a plan's future success, Eq. 3.12, in two parts. The probability that the plan will succeed is reduced to the probability that the plan succeeded by simulating the plant. The probability that the plan succeeded depends on testing if state trajectories are accepted by the plan. We assumed an estimator capable of computing $\mathbf{P}\left(\mathbf{o}^i|B_P^i\right)$, B^i, B_P^i and $\mathbf{P}\left(\mathbf{m}^{i+1}|\mathbf{m}^i,\boldsymbol{\mu}^{i+1}\right)$.

The next chapter presents an approach to approximating these equations based on sampling from both the observation and transition sequences in order to bound the number of terms considered.

Chapter 4

Task Monitoring Algorithm

This chapter develops an algorithm that approximates the probability that the plan will succeed, Eq. 3.12 repeated here:

$$\mathbf{P}\left(QSP_{succ}|\mathbf{o}^{1:t},\boldsymbol{\mu}^{1:t}\right) = \sum_{\mathbf{o}^{t+1:n}\in\mathbb{D}_{\mathbf{O}^{t+1:n}}} \prod_{i=t+1}^{n} \mathbf{P}\left(\mathbf{o}^{i}|B_P^{i}\right)\cdot\mathbf{P}\left(QSP_{succ}|\mathbf{o}^{1:n},\boldsymbol{\mu}^{1:n}\right) \tag{4.1}$$

$$\mathbf{P}\left(QSP_{succ}|\mathbf{o}^{1:n},\boldsymbol{\mu}^{1:n}\right) = \sum_{\mathbf{m}^{0:n}\in\text{Trajs(QSP)}} \left(\prod_{i=0}^{n-1} B^{i}\left(\mathbf{m}^{i}\right)\frac{\mathbf{P}\left(\mathbf{m}^{i+1}|\mathbf{m}^{i},\boldsymbol{\mu}^{i+1}\right)}{B_P^{i+1}\left(\mathbf{m}^{i+1}\right)}\right) B^{n}\left(\mathbf{m}^{n}\right) \tag{4.2}$$

by sampling the possible future events, the observations and paths, of the system of interest. We first present the general algorithm for computing Eq. 3.12 as a depth-first computation that enumerates all events. We then reformulate the algorithm into one that makes a series of random choices and then it tests the sample generated to see if it is a member of the trajectories accepted by the QSP.

For the purpose of this chapter, we assume an estimator for the PCCA system model that can update the belief state and compute the probability of observations. These are both presented in Chapter 7. This work also assumes we have an algorithm to map the current objective of the QSP into a set of commands to be issued to the hardware. The QSP may be labeled with the necessary commands, requiring a no-op algorithm, or, alternatively, the QSP may just be labeled with the desired partial state, and a reactive planner can be used to generate the commands necessary to move the system from the current state to a state that matches the desired partial state, such as

Chung [8].

4.1 Explicit Evaluation

This section describes an algorithm that enumerates all events, computing a very precise estimate that the plan will succeed. The algorithm in this section is only approximating the belief state by examining the k most probable states. This section is broken into two parts. The first part is the PlanWillSucceed algorithm that predicts the future success of the plan, $\mathbf{P}\left(QSP_{succ}|\mathbf{o}^{1:t}, \boldsymbol{\mu}^{1:t}\right)$. The second part is the PlanSucceeded algorithm that predicts the probability that a particular complete execution of the plan actually resulted in a successful plan execution, $\mathbf{P}\left(QSP_{succ}|\mathbf{o}^{1:n}, \boldsymbol{\mu}^{1:n}\right)$. Since we are reasoning about hidden states, there is always some chance that the plant only appeared to successfully execute the plan, but in fact did not.

Consider a simple example. We have a simple switch, which for the purpose of this example is either on or off. In this example, the switch starts off with probability 1. Assume the QSP requires that the switch be on in exactly one step. Also assume our controller will issue a command $\boldsymbol{\mu}^{t+1}$ to turn on the switch with the probabilities:

$$\begin{aligned}\mathbf{P}\left(\mathrm{on}^{t+1} \,|\, \mathrm{off}^{t}, \boldsymbol{\mu}^{t+1}\right) =& 0.9\\ \mathbf{P}\left(\mathrm{off}^{t+1} \,|\, \mathrm{off}^{t}, \boldsymbol{\mu}^{t+1}\right) =& 0.1\end{aligned}$$

Our model has the observation function:

$$\begin{aligned}\mathbf{P}\left(\mathbf{o}_1^t|\,\mathrm{on}^t\right) =& \quad 0.9 \quad \mathbf{P}\left(\mathbf{o}_2^t|\,\mathrm{on}^t\right) = \quad 0.1\\ \mathbf{P}\left(\mathbf{o}_1^t|\,\mathrm{off}^t\right) =& \quad 0.1 \quad \mathbf{P}\left(\mathbf{o}_2^t|\,\mathrm{off}^t\right) = \quad 0.9\end{aligned}$$

For this example, if time starts at $t = 0$, then our plan always has length $n = 1$, so Eq. 4.1 simplifies to a summation over the possible observations at time 1. The possible observations are $\mathbf{o}_1^1$ and $\mathbf{o}_2^1$. The predicted belief B_P^1 is 0.9 that on^1 is the state and 0.1 that off^1 is the state. Thus,

for the observation $\mathbf{o}_1^1$, we can evaluate:

$$\begin{aligned}\mathbf{P}\left(\mathbf{o}_1^1|B_P^1\right) &= \mathbf{P}\left(\mathbf{o}_1^1|\,\mathrm{on}^1\right) \times B_P^1\left(\mathrm{on}^1\right) + \mathbf{P}\left(\mathbf{o}_1^1|\,\mathrm{off}^1\right) \times B_P^1\left(\mathrm{off}^1\right) \\ &= 0.9 \times 0.9 + 0.1 \times 0.1 \\ &= 0.82\end{aligned}$$

The estimated belief state B^1 given $\mathbf{o}_1^1$, from the estimator, is 0.988 that on^1 is the state and 0.012 that off^1 is the state. For the observation $\mathbf{o}_1^1$ and each path $\mathbf{m}^{0:1}$ that achieves that QSP, Eq. 4.2 evaluates

$$B^0\left(\mathbf{m}^0\right)\frac{\mathbf{P}\left(\mathbf{m}^1|\mathbf{m}^0,\boldsymbol{\mu}^1\right)}{B_P^1\left(\mathbf{m}^1\right)}B^1\left(\mathbf{m}^1\right) \tag{4.3}$$

For our example with $\mathbf{o}_1^1$, there are two possible paths, $\left\{\mathrm{off}^0,\mathrm{off}^1\right\}$ and $\left\{\mathrm{off}^0,\mathrm{on}^1\right\}$. Only the path $\left\{\mathrm{off}^0,\mathrm{on}^1\right\}$ achieves the QSP goal of being on at time 1. So evaluating Eq. 4.3 on $\left\{\mathrm{off}^0,\mathrm{on}^1\right\}$ yields:

$$\begin{aligned}\mathbf{P}\left(QSP_{succ}|\left\{\mathbf{o}_1^1\right\},\left\{\boldsymbol{\mu}^1\right\}\right) &= B^0\left(\mathrm{off}^0\right)\frac{\mathbf{P}\left(\mathrm{on}^1\,|\,\mathrm{off}^0,\boldsymbol{\mu}^1\right)}{B_P^1\left(\mathrm{on}^1\right)}B^1\left(\mathrm{on}^1\right) \\ &= 1\frac{0.9}{0.9}0.988 \qquad\qquad = 0.988\end{aligned}$$

Backing out to the $\mathbf{P}\left(QSP_{succ}|\mathbf{o}^{1:t},\boldsymbol{\mu}^{1:t}\right)$ computation, we have now computed that the overall probability that the plan will succeed for the $\mathbf{o}_1^1$ option is $0.988 * 0.82 = 0.81$. For the other observation $\mathbf{o}_2^1$, the same terms can be calculated:

$$\begin{aligned}\mathbf{P}\left(\mathbf{o}_2^1|B_P^1\right) &= \mathbf{P}\left(\mathbf{o}_2^1|\,\mathrm{on}^1\right) \times B_P^1\left(\mathrm{on}^1\right) + \mathbf{P}\left(\mathbf{o}_1^1|\,\mathrm{off}^1\right) \times B_P^1\left(\mathrm{off}^1\right) \\ &= 0.1 \times 0.9 + 0.9 \times 0.1 \\ &= 0.18\end{aligned}$$

$$
\begin{aligned}
\mathbf{P}\left(QSP_{succ}|\left\{\mathbf{o}_2^1\right\},\left\{\boldsymbol{\mu}^1\right\}\right) &= B^0\left(\mathrm{off}^0\right)\frac{\mathbf{P}\left(\mathrm{on}^1\,|\,\mathrm{off}^0,\boldsymbol{\mu}^1\right)}{B_P^1\left(\mathrm{on}^1\right)}B^1\left(\mathrm{on}^1\right)\\
&= 1\frac{0.9}{0.9}0.5\\
&= 0.5
\end{aligned}
$$

And thus the probability of success given the $\mathbf{o}_2^1$ option is $0.18 * 0.5 = 0.09$. The total probability $\mathbf{P}\left(QSP_{succ}|\mathbf{o}^{1:t},\boldsymbol{\mu}^{1:t}\right) = 0.81 + 0.9 = 0.9$. The value 0.9 should be expected as this represents the predicted probability of being in the on^1 state. In this simple example, the controller is only allowed one action and its taken prior to any of our simulated information, so it cannot do better than the simple predicted outcome. If, for instance, the QSP allowed 2 steps to achieve on instead of just one, the controller could re-issue the turn-on command if off^1 is observed and do better that the predicted 0.9.

We now present the explicit algorithm for computing the probability that the plan will succeed $\mathbf{P}\left(QSP_{succ}|\mathbf{o}^{1:t},\boldsymbol{\mu}^{1:t}\right)$ explicitly.

4.1.1 Plan Will Succeed

The algorithm PlanWillSucceed (PWS), Alg. 4.1, computes Eq. 3.11 and uses the algorithm PlanSucceeded (PS), Alg. 4.2, that computes Eq. 3.9 as a sub-routine. Recall that Eq. 3.9 evaluates $\mathbf{P}\left(QSP_{succ}|\mathbf{o}^{1:n},\boldsymbol{\mu}^{1:n}\right)$. All algorithms depend on the QSP and PCCA system model. We repeat Eq. 3.11 here:

$$
\mathbf{P}\left(QSP_{succ}|\mathbf{o}^{1:t},\boldsymbol{\mu}^{1:t}\right) = \sum_{\mathbf{o}^{t+1:n}\in\mathbb{D}_{\mathbf{O}^{t+1:n}}}\prod_{i=t+1}^{n}\mathbf{P}\left(\mathbf{o}^i|B_P^i\right)\cdot\mathbf{P}\left(QSP_{succ}|\mathbf{o}^{1:n},\boldsymbol{\mu}^{1:n}\right).
$$

The algorithm works by simulating the controller, which maps the previous state into a command using the plan, and then considering all observations that are consistent with that command. Since the controller is actually two algorithms, a scheduler that updates the QSP based on the estimated state sequence and a reactive planner that maps the current partial schedule into a command,

Algorithm 4.1: PlanWillSucceed(M_{QSP}, $B^{0:t-1}$, $B_P^{1:t}$, B^t, $\mathbf{o}^{1:t}$, $\boldsymbol{\mu}^{1:t}$)

```
1  M'_QSP ← M_QSP updated given B^t ;
2  if M'_QSP indicates the plan is done executing then
3      n ← t ;
4      return PlanSucceeded(B^{0:n}, B_P^{1:n}, o^{1:n}, μ^{1:n}, {}) ;
5  end
6  μ^{t+1} ← Best command to issue next, given M'_QSP and B^i ;
7  B_P^{t+1} ← Belief state estimate given μ^{t+1} ;
8  p ← 0 ;
9  forall o^{t+1} do
10     p_o ← P(o^{t+1} | B_P^{t+1}) ;
11     B^{t+1} ← Belief state estimate given μ^{t+1}, o^{t+1} ;
12     p_o ← p_o · PlanWillSucceed(M'_QSP, B^{0:t}, B_P^{1:t+1}, B^{t+1}, o^{1:t+1}, μ^{1:t+1}) ;
13     p ← p + p_o ;
14 end
15 return p ;
```

we simulate these two parts separately in our algorithm. We treat these algorithms separately because the scheduler lets us know when it thinks the plan is done (or failed). The rest of this PWS algorithm is computing Eq. 3.11 given this command.

This algorithm takes the parameters:

- M_{QSP} – The current schedule in the QSP. The current schedule specifies only the times at which past events occurred and thus also the current activities. The schedule can be used to determine if the plan has completed (or failed) and what the next step is in the plan.

- $B^{0:t-1}$, B^t – The belief state estimates over the course of the whole plan thus far, from time 0 to t. We need the past belief states for the algorithm PS.

- $B_P^{1:t}$ – The predicted belief state estimates over the course of the whole plan thus far, from time 1 to t. We need these for the algorithm PS.

- $\mathbf{o}^{1:t}$, $\boldsymbol{\mu}^{1:t}$ – The current accumulated list of observations and commands, respectively. These are extended at every step of this algorithm by one additional term for use by the algorithm PS.

On Line 1, the PWS algorithm starts by updating the schedule based on the current belief state. We use the dispatcher algorithm FAST-DC [52] to update the schedule. FAST-DC schedules the events in the plan in a greedy fashion based on the current most likely state, subject to the time bounds of the activities. We assume that the dispatcher is deterministic, as is the case with FAST-DC.

On lines 2-5, the algorithm looks for the end of the plan, either because the final event was fixed or because the plan failed. In the case of plan failure, we expect in general that PS will return 0, but there is some chance that the plant successfully executed the plan despite the dispatcher's indication that the plan failed. This non-zero case usually arises near the end of a plan when the most likely state causes the plan to fail but some less likely state achieves the plan. Within these lines of the algorithm, Line 3 notes that the dispatcher believes the plan completed at time n. On Line 4 we call the PS algorithm given the initial belief state and the commands and observations generated up through the final time point n.

Should the plan not be complete yet, the algorithm simulates the system's model forward one time interval. The first step to simulating the system is to determine the action that the executive will issue, given the current plan's progress and the belief state of the plant, as done by Line 6. This command may be labeled on the plan or we may use a planner such as [8] to determine this command, depending on how the actual executive is configured. Again, we assume this command is deterministic.

The algorithm then uses the estimation algorithm without any observations to predict the next belief state on Line 7. This predicted distribution B_P^{t+1} allows us to determine the relative likelihood of each observation. The algorithm then sets our probability that the plan will succeed p to 0 on Line 8.

On lines 9-14, the algorithm enumerates all of the possible observations at time $t+1$, computing the probability that each observation will lead to a successful plan execution, when included with the current observation sequence. Line 10 uses the estimator to compute the probability of the observation chosen in this iteration of the loop. Line 11 uses the estimator to update the belief state given this observation. Line 12 then recursively calls PWS on this simulated system state, namely

the new belief state and the command and observation sequences extended to include the chosen command and observation, respectively. The probability returned by PWS is multiplied by the probability of the observation, giving the total probability that the observation leads to a successful plan execution. On Line 13, the algorithm adds this total probability to the accumulated probability p that the plan will succeed from time t.

Once all the observations have been considered, the algorithm returns p, the probability that the plan will succeed given it has executed up to time t, on Line 15.

Note that PWS is able to compute the probability that the plan will succeed starting from any time t, based on the actual plan execution data. If $t = 0$, then PWS computes the probability the plan will succeed from an initial belief B^0.

4.1.2 Plan Succeeded

Algorithm 4.2: $\mathrm{PS}\big(B^{0:n}, B_P^{1:n}, \mathbf{o}^{1:n}, \boldsymbol{\mu}^{1:n}, \mathbf{m}^{t+1:n}\big)$

```
1  p ← 0 ;
2  forall m^t do
3      if t = n then
4          p_o ← B^t (m^t) ;
5      else
6          p_τ ← P (m^{t+1} | m^t, μ^t) ;
7          p_o ← B^t (m^t) · p_τ / B_P^{t+1} (m^{t+1}) ;
8      end
9      if t = 0 then
10         if Traj (m^{0:n}) then
11             p ← p + p_o ;
12         end
13     else
14         p_o ← p_o · PS(B^{0:n}, B_P^{1:n}, o^{1:n}, μ^{1:n}, m^{t:n}) ;
15         p ← p + p_o ;
16     end
17 end
18 return p ;
```

The algorithm PlanSucceeded (PS), Alg. 4.2 computes Eq. 3.9, reprinted here:

$$\begin{aligned}\mathbf{P}\left(QSP_{succ}|\mathbf{o}^{1:n},\boldsymbol{\mu}^{1:n}\right) &= \sum_{\mathbf{m}^{0:n}\in\text{Trajs(QSP)}} \mathbf{P}(\mathbf{m}^{0:n}|\mathbf{o}^{1:n},\boldsymbol{\mu}^{1:n}) \\ &= \sum_{\mathbf{m}^{0:n}\in\text{Trajs(QSP)}} \left(\prod_{i=0}^{n-1} B^i\left(\mathbf{m}^i\right) \frac{\mathbf{P}\left(\mathbf{m}^{i+1}|\mathbf{m}^i,\boldsymbol{\mu}^{i+1}\right)}{B_P^{i+1}\left(\mathbf{m}^{i+1}\right)}\right) B^n\left(\mathbf{m}^n\right)\end{aligned}$$

The PS algorithm uses Eq. 3.6 to recursively compute extensions to the current trajectory $\mathbf{m}^{t+1:n}$ backwards in time until it has a trajectory that spans the observation and command data. It then uses Traj to determine if the trajectory is a member of the trajectories accepted by the QSP. The details of the Traj function are at the end of this section. The algorithm adds $\mathbf{P}(\mathbf{m}^{0:n}|\mathbf{o}^{1:n},\boldsymbol{\mu}^{1:n})$ to the total probability that the plan succeeded if Traj accepts the trajectory.

When initially called by the PWS algorithm, the PS algorithm has an empty trajectory, denoted $\mathbf{m}^{n+1:n}$. The first iteration thus chooses a starting point for the trajectory from B^n, executing Line 4. Further iterations extend the trajectory towards $t = 0$ based on the observations and commands.

The algorithm begins by setting the probability the plan succeeded to 0 on Line 1. The lines 2-17 loop over all of the states with non-zero belief at time t. These are each considered in turn as extensions to the current trajectory $\mathbf{m}^{t+1:n}$.

Line 6 computes the probability of transitioning from the state $\mathbf{m}^t$ to the given state $\mathbf{m}^{t+1}$, subject to the command issued. This is computed by the estimation algorithm and is a part of the belief state estimation computation. Line 7 computes the smoothed probability of being in state $\mathbf{m}^t$ given that the next state is $\mathbf{m}^{t+1}$ based on all the observations and commands, the non-recursive part of Eq. 3.6.

If the trajectory is completely specified from time 0 to n, i.e. when $t = 0$, then the algorithm, on lines 9-13, evaluates the trajectory. Otherwise, the algorithm recursively considers extensions from the state $\mathbf{m}^t$ on lines 13-16.

For a complete trajectory $\mathbf{m}^{0:n}$, the algorithm tests if the trajectory is accepted by Traj as a trajectory that achieves the plan on Line 10. If the trajectory achieves the plan, its incremental probability is accumulated into p, as specified by p_0, on Line 11. Otherwise, the trajectory and its

associated probability do not succeed so it is not added to p.

Line 14 recursively computes the probability that a trajectory ending with $\mathbf{m}^{t:n}$ achieves the plan. The resulting product of the probability of being in $\mathbf{m}^{t}$ given $\mathbf{m}^{t+1:n}$ and the probability that it leads to an accepted trajectory is accumulated on Line 15 into p.

The algorithm returns the probability that the plan succeeded on Line 18.

Traj function

The Traj function is responsible for determining if the trajectory $\mathbf{m}^{0:n}$ is a member of the trajectories accepted by the QSP. The approach we use for this sub-problem is to frame the problem as a temporal plan network (TPN) problem [58]. A TPN is a superset of our QSP and the TPN supports one additional feature relevant to this thesis: we can label activities with any number of Ask (A) and Tell (A) constraints, where A is some assignment to a set of TPN variables. A solution to a TPN is temporally flexible schedule, an STN-u, such that all activities with Ask constraints are constrained to be temporally contained within activities with matching Tell constraints.

The idea is that if we label our QSP to *ask* for the states it requires at each event and label the trajectory to *tell* the actual states, then we can use the planner Kirk [6, 24] to see if there exists a temporally consistent schedule for this proposed TPN. If a schedule exists, then the trajectory achieves the plan.

For the QSP part, we construct a TPN from the QSP such that every event ev in the QSP has a corresponding event in the TPN with the same temporal constraints. We add to each of these events a second event and a constraint between the two events that requires them to occur at the same instant. We then label this new constraint with an Ask $(\boldsymbol{\sigma}_{goal})$, where the $\boldsymbol{\sigma}_{goal}$ is the partial state of the activity with ev as its finish event.

For the trajectories, we create a chain of events for each variable. The first event occurs half a step before the plan starts and we add additional events each time the state changes along with an end event. For each interval, we label the interval with a *tell* constraint that specifies the state along that interval. We place the events half a step displaced earlier than the events of the QSP since we want the instantaneous *ask* events of the QSP to fall within the *tell* event of the state, and so the half

step offset ensures that the *ask* event can occur at the onset of the desired state.

For example, Fig. 4-1 is the result of adding *ask* and *tell* constraints to the example shown in Fig. 3-3 on page 43. Each state required by the plan in Fig. 3-3 has been replaced by a pair of events and an episode with an *ask* constraint that requires the same state be true between the two new events in Fig. 4-1. Each trajectory is likewise replaced by a pair of events that capture each interval over which a state is constant. Each pair of events has a new episode with a *tell* constraint that asserts the actual state during that interval.

Once we have constructed this TPN, we can ask a TPN planner, such as [24, 35], if there exists a solution to this TPN. If there does, then Traj returns true. Otherwise, the planner is unable to fix the QSP events so they are consistent with the state trajectory, and so we return false.

For example, in Fig. 4-1, the planner states that the example is consistent, and is specifically consistent with the addition of the dashed-line episodes. These new episodes ensure that each *ask* constraint is contained within a matching *tell* constraint.

Summary

In this section we have presented a pair of algorithms for computing the probability that a plan will succeed and the probability the plan did succeed. The first algorithm has a branching factor of $|\mathbf{o}|$ and a depth of n and the second algorithm has a branching factor of $|\mathbf{m}|$ and a depth of $n+1$. Together these algorithms make $|\mathbf{o}|^n \cdot |\mathbf{m}|^{n+1}$ decisions. Since we expect both large branching factors and large n, the next section presents methods of approximating this probability using samples.

4.2 Sampling Task Monitoring Equations

In this section we show how to frame Algs. 4.1 and 4.2 as a Monte Carlo [48] sampling problem that approximates the same problem with a user specified number of samples. We can also frame the sampling problem as an anytime algorithm that generates an approximation for the probability that the plan will succeed after a fixed time interval, so long as the algorithm has enough time to take one sample. The accuracy of the answer improves as more samples are taken and time is allotted. We use a sampling-based approach as our problem has the special property that the choice to continue

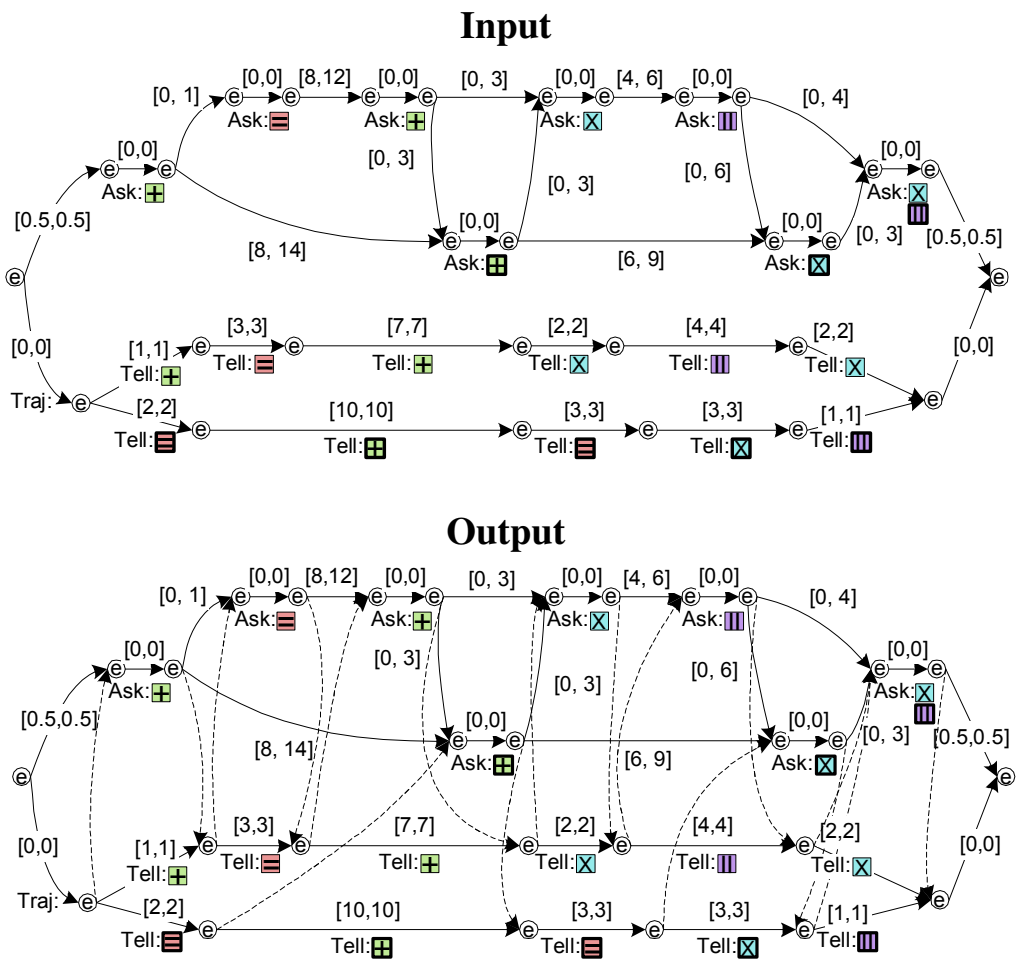

Figure 4-1: This figure shows an example input for the Traj function and its corresponding output. The input is generated by adding *ask* and *tell* constraints to Fig. 3-3. The Traj function adds the dashed lines to the TPN in order to satisfy the *ask* and *tell* constraints. The states of the second variable are shown with a thick box around them while the states for the first variable have thin-lined boxes. The dashed lines indicate within which *tell* episode each *ask* episode can be placed in order to satisfy both the temporal constraints as well as the *ask* and *tell* constraints.

or discontinue the plan is what this probability is being used for, and this decision requires only a coarse likelihood of future success.

For example, if we only need to know the probability of success within a 5% standard deviation, we only need 400 samples. This number of samples is independent of the complexity of the plan, though of course more complex plans require more work to generate a single sample. For our problem, the sample generating algorithm scales linearly with the length of the plan, for the same plant model.

4.2.1 Monte Carlo

We can reduce the number of paths that need to be examined by employing Monte Carlo sampling techniques, as presented in [48]. Monte Carlo sampling involves choosing values for a random variable based on its distribution. For instance, consider a coin random variable with the two values *heads* and *tails*. If we assume the coin is fair, and thus there is a 50-50 chance of each value, then a sample of the coin will turn up heads and tails about as often.

Monte Carlo sampling is an effective way to estimate a probability distribution with an accuracy that improves based on the number of samples taken. The standard deviation of the error is proportional to $\frac{1}{\sqrt{n}}$, where n is the number of samples. In our problem, we are interested in computing the probability that the plan will succeed or not, so our random variable, like the coin, only has two outcomes. We next show that we can approximate the distributions we want to sample, as is required by this approach.

4.2.2 Sampling Algorithms

We now present the re-formulated versions of Algs. 4.1 and 4.2 using Monte Carlo Sampling, Algs. 4.3 and 4.4, respectively. In Section 4.2.3, we present the sub-routines used to sample from the distributions presented here.

These sampling algorithms return *true* if the generated sample achieves the plan. We assume that the calling algorithm counts the number of samples that are accepted vs. not accepted. The estimated likelihood that the plan will succeed is the ratio of the number of accepted samples over

the total number of samples: $\frac{\#\text{true}}{\#\text{samples}}$. Each invocation of the algorithm uses the same input. We show in Section 4.2.4 that we only make two approximations to the distribution: the estimator prunes the belief state to k elements at each step of the prediction and thus the trajectory sampling code is also limited to these k states at each step. These two values of k need not be coupled, but using larger values of k for trajectories requires that the system be re-estimated at the higher values of k. For sufficiently large k, the prediction's accuracy converges to the true probability as the number of samples goes to infinity.

Returning to our switch example from Section 4.1, recall that the switch is either on or off. For the example, the switch starts off and the QSP specifies that it should be on in one step. Recall that there were two possible observations $\mathbf{o}_1^1$ and $\mathbf{o}_2^1$, and two possible paths $\left\{\text{off}^0, \text{off}^1\right\}$ and $\left\{\text{off}^0, \text{on}^1\right\}$. The probability of $\mathbf{o}_1^1$ is 0.82 and the probability of $\mathbf{o}_2^1$ is 0.18. For $\mathbf{o}_1^1$, the probability of the paths are 0.012 and 0.988, respectively. For $\mathbf{o}_2^1$, the probability of the paths are both 0.5.

For the sampling approach, we might generate these ten (grouped) samples:

#	Sample	Prob of Sample
8	$\mathbf{o}_1^1, \left\{\text{off}^0, \text{on}^1\right\}$	0.81
1	$\mathbf{o}_2^1, \left\{\text{off}^0, \text{on}^1\right\}$	0.09
1	$\mathbf{o}_2^1, \left\{\text{off}^0, \text{off}^1\right\}$	0.09

The first two samples accomplish the QSP, so $\text{Traj}\left(\left\{\text{off}^0, \text{on}^1\right\}\right)$ returns *true* while the last sample does not. After ten (high-quality) samples, our estimated probability that the plan will succeed is $\frac{8+1}{10} = 0.9$. For this example, this is also the true probability. The omitted sample $\left\{\mathbf{o}_1^1, \left\{\text{off}^0, \text{off}^1\right\}\right\}$ has about a one percent chance of occurring. We now present the algorithms that generate the samples illustrated in this example.

Plan Will Succeed – Sampled

In Alg. 4.3, we have replaced the loop over all $\mathbf{o}^{t+1}$ from Alg. 4.1 with a routine that samples from $\mathbf{P}\left(\mathbf{o}^{t+1} | B_P^{t+1}\right)$. We discuss this sampling routine in Section 4.2.4. The other major change is that this algorithm is no longer accumulating the probability that the plan will succeed, it is instead

Algorithm 4.3: PWSSamp(M_{QSP}, $B^{0:t-1}$, B^t, $\mathbf{o}^{1:t}$, $\boldsymbol{\mu}^{1:t}$)

1 $M'_{\text{QSP}} \leftarrow M_{\text{QSP}}$ updated given B^t ;
2 **if** M'_{QSP} *indicates the plan is done executing* **then**
3 $\quad n \leftarrow t$;
4 $\quad$ **return** *PSSamp*$\left(B^0, \mathbf{o}^{1:n}, \boldsymbol{\mu}^{1:n}, \{\}\right)$;
5 **end**
6 $\boldsymbol{\mu}^{t+1} \leftarrow$ Best command to issue next, given M'_{QSP} and B^i ;
7 $B_P^{t+1} \leftarrow$ Belief state estimate given $\boldsymbol{\mu}^{t+1}$;
8 $\mathbf{o}^{t+1} \leftarrow$ Sample from $\mathbf{P}\left(\mathbf{o}^{t+1} | B_P^{t+1}\right)$;
9 $B^{t+1} \leftarrow$ Belief state estimate given $\boldsymbol{\mu}^{t+1}, \mathbf{o}^{t+1}$;
10 **return** *PWSSamp*$\left(M'_{QSP}, B^{0:t}, B^{t+1}, \mathbf{o}^{1:t+1}, \boldsymbol{\mu}^{1:t+1}\right)$;

testing if the sampled observation leads to a successful execution of the plan. The call to PSSamp on Line 4 returns *true* if the trajectory sampled from the observation and command sequence achieved the plan's objectives.

Lines 1-7 are identical in Algs. 4.3 and 4.1. In these lines the algorithm simulates the dispatcher and calls PSSamp if the plan is done. The algorithm then determines the command that would be issued next and predicts the next belief state using this command. On Line 8, the algorithm samples from the observation probability given this predicted belief state. The algorithm then updates the belief state given this sampled observation on Line 9 and recursively calls PWSSamp given this sampled observation and next belief state, returning the result on Line 10.

Plan Succeeded – Sampled

The PSSamp algorithm returns true if the sampled trajectory is consistent with Traj and false otherwise. For the first iteration, when $t = n$, the PSSamp algorithm samples from the final belief state B^n on Line 2. Otherwise, the PSSamp algorithm begins on lines 4-7 by computing the smoothed distribution B_*^t for each non-zero $\mathbf{m}^t$, corresponding to lines 6 and 7 of the PS algorithm. Since $B_P^{t+1}\left(\mathbf{m}^{t+1}\right)$ on Line 7 of Alg. PS is constant for all $\mathbf{m}^t$ and thus does not effect our sampling distribution, we omit it from the calculation on Line 6 of Alg. PSSamp. Line 9 then samples from B_*^t using the routine describe in Section 4.2.3.

In Alg. 4.4, the same base case applies as in Alg. 4.2, namely when $t = 0$, the algorithm

Algorithm 4.4: PSSamp$\left(B^{0:n}, \mathbf{o}^{1:n}, \boldsymbol{\mu}^{1:n}, \mathbf{m}^{t+1:n}\right)$

```
1  if t = n then
2      B*^t = B^t ;
3  else
4      forall m^t do
5          p_τ ← P(m^{t+1} | m^t, μ^t) ;
6          B*^t(m^t) = B^t(m^t) · p_τ ;
7      end
8  end
9  m^t ← Sample from B*^t ;
10 if t = 0 then
11     return Traj(m^{0:n}) = true ;
12 else
13     return PSSamp(B^{0:n}, o^{1:n}, μ^{1:n}, m^{t:n}) ;
14 end
```

has sampled a full-length trajectory and can test this using Traj. Line 11 returns whether or not the trajectory $\mathbf{m}^{0:t}$ is accepted by Traj. If the trajectory is not yet at full length, the algorithm recursively calls itself on Line 13.

4.2.3 Belief States

Algorithm 4.5: SampleBeliefState$\left(B^t\right)$

```
A ← {⟨m^t, p⟩ | p = B^t(m^t)} ;
r ← Random value ∈ [0, 1] ;
i ← 0 ;
while A[i].p < r do
    r ← r − A[i].p ;
    i ← i + 1 ;
end
return A[i].m^t ;
```

We start with the least complicated sampling algorithm, that of sampling from B^t, on Line 9 of Alg. 4.4. Recall that B^t is a function that maps a state $\mathbf{m}^t$ to a probability, so the belief state is an explicit discrete probability density function. As we show in Section 7.3, we approximate B^t by

the k most probable states. To sample from this function we can convert it into an array of entries, where each entry is a state $\mathbf{m}^t$ paired with the state's probability $B^t\left(\mathbf{m}^t\right)$. We only store states with non-zero probabilities. For example, this may generate the array $\left[\,\langle x_1, 0.1\rangle\,,\,\langle x_2, 0.5\rangle\,,\,\langle x_3, 0.3\rangle\,,\,\langle x_4, 0.1\rangle\,\right]$. We next choose a random number r between 0 and 1, and then walk through the array looking for the $\mathbf{m}^t$ that corresponds to this probability in the density function. We can find this item by sequentially walking through the array and subtracting the density function from our value r until we find an entry that makes r non-positive. This is implemented by Alg. 4.5.

Note that we could pre-integrate the values and do a binary search, but we expect to sample from B^t exactly once, so this is not advantageous to our problem.

4.2.4 Observations

In order to sample from $\mathbf{P}\left(\mathbf{o}^{t+1}|B_P^{t+1}\right)$, we can use OCSP solver to extract the k most probable observations. The mapping from the extraction of the k most probable observations to the OCSP solver of chapter 5 is shown in Section 7.5. By extracting the k most probable observations, we can use the same Alg. 4.5 above for sampling from the belief states.

We can, however, improve upon this approach by taking advantage of the internal representation used in Chap. 5 to solve this OCSP. In Section 5.6, we provide an algorithm DNNFSample, Alg. 5.3, that can sample from the distribution computed on the representation that the OCSP solver uses to extract the k best solutions. Thus, if we use the OCSP solver to compile the computation of $\mathbf{P}\left(\mathbf{o}^{t+1}|B_P^{t+1}\right)$ into its internal representation, we can use Alg. 5.3 on Line 8 of Alg. 4.3 to sample observations.

Changing to using Alg. 5.3 improves upon using Alg. 4.5 in two ways: a reduction in algorithm complexity and an increase in fidelity. Alg. 4.5 has a complexity dominated by Alg. 6.1, which extracts the k best observations. Alg. 6.1's time complexity is approximately $O(|E|k\log k)$, where E is the number of edges of the OCSP representation. Alg. 5.3 is dominated by the steps taken by Alg. 5.1, which computes the contribution of each of the k belief states. Alg. 5.1's time complexity is only $O(|E|k)$.

With respect to fidelity, Alg. 4.5 only samples from the k most probable observations, which

we explicitly enumerate. Alg. 6.1 samples from all the solutions using the factored representation. Thus, Alg. 6.1 is sampling from the actual distribution, given the k best state approximation made by B_P^{t+1}.

4.2.5 Runtime Analysis

We now analyze the complexity of PWSSamp (Alg. 4.3) and PSSamp (Alg. 4.4). We start with PSSamp. This algorithm recurses $(n+1)$ times before evaluating Traj. At each step t it samples $\mathbf{m}^t$, where this is using SampleBeliefState (Alg. 4.5). SampleBeliefState has an $O(k)$ time and space complexity, where k the size of B^t. Evaluating the transition probability on Line 5 has a complexity that is linear in the size of the estimator's representation e, which is the size of the compiled PCCA model. PSSamp calls Traj exactly once, where its complexity depends on the implementation chosen and the size of the plan. Together, the algorithm has a complexity of $O(ne+nk)$ plus the complexity of Traj.

The algorithm PWSSamp, Alg. 4.3 recurses up to n times before calling PSSamp once. It invokes the dispatcher and the command generation code once per recursion. It uses Accumulate and DNNFSample once per recursion to sample an observation. If we let the size of the OCSP representation for the observation sampling be e_o, then each invocation of Accumulate and DNNFSample has a time complexity of $O(e_o k)$. The algorithm also invokes the estimation algorithm twice, with a complexity of $O(ek \log k)$ both times. If we ignore PSSamp for now, this algorithm has a time complexity of $O(ne_o k + nek \log k)$ plus n times the complexity of the dispatcher and command generation code. We note that $e_o < e$ in general since the constraints and variables of e_o are a subset of those used to generate e. This simplifies our complexity to $O(nek \log k)$, which dominates the $O(ne + nk)$ time complexity of the PSSamp algorithm. Thus, the complexity of PWSSamp is at least $O(nek \log k)$, plus the complexity of n invocations of the dispatcher and command generation code, and the single invocation of Traj.

Note that the PWSSamp algorithm's complexity is the complexity required to run a plan on the actual embedded device over n steps, $O(nek \log k)$, plus the $O(ne_o k)$ complexity of sampling an observation and the $O(ne + nk)$ and Traj complexity of sampling a trajectory, all of which are less

complex than the system itself. Thus, for plans of a modest length n and for a modest number of samples, this is expected to be tractable.

4.3 Conclusion

This chapter has shown how to generate a sample that can be tested to see if it meets the requirements of the QSP. By accumulating the number of samples accepted by the plan, we can approximate the probability that the plan will succeed. Since this function can be invoked at any point during the plan execution, this probability can be computed using PWSSamp (Alg. 4.3) at any point. This algorithm has a complexity that is linear in the length of the plan being executed and has a proportional complexity to the number of estimates being tracked k, specifically $(k \log k)$ and the size of the compiled representation e. The next two chapters show how to solve the optimal constraint satisfaction problem used in this chapter for both estimation and sampling. Chapter 7 then presents the derivation of the estimation and observation equations and how they map to the OCSPs used in this chapter and solved in the next chapter.

Chapter 5

Optimal Constraint Satisfaction Problem Compilation

This chapter explains how to compile and solve a class of Optimal Constraint Satisfaction Problems (OCSP) [59, 50] that arise in this thesis. An OCSP is related to a Valued Constraint Satisfaction Problem (VCSP) [51, 5], except all valuations on variable assignments are unary. This class of OCSPs is defined over a set of variables $\mathbf{X}$, which are partitioned into three sets: $\mathbf{X}_M$, $\mathbf{X}_\Sigma$, and $\mathbf{X}_R$. We denote assignments to these variables in lower case: $\mathbf{x}$. Given an assignment to $\mathbf{X}_a \subset \mathbf{X}$, typically $\mathbf{X}_a \subset \mathbf{X}_R$, we are interested in compiling and solving the expression:

$$\underset{\mathbf{x}_{M\setminus a} \in \mathbb{D}_{\mathbf{X}_M \setminus \mathbf{X}_a}}{\arg^{\mathrm{k}}\max} \sum_{\mathbf{x}_{\Sigma\setminus a} \in \mathbb{D}_{\mathbf{X}_\Sigma \setminus \mathbf{X}_a}} \left(\frac{\prod_{i=1}^{m} g_i\left(\mathbf{x}_{M\setminus a} \cup \mathbf{x}_{\Sigma\setminus a} \cup \mathbf{x}_a\right)}{\prod_{i=m+1}^{n} g_i\left(\mathbf{x}_{M\setminus a} \cup \mathbf{x}_{\Sigma\setminus a} \cup \mathbf{x}_a\right)} \cdot \alpha_{\mathbf{x}_\Sigma} \right) \tag{5.1}$$

Expressions of this form arise in all of our estimation computations. One of which, for example, is approximating the belief state with the k best states, given the k best previous states. In this example, the next states are $\mathbf{x}_M$ and the previous states are $\mathbf{x}_\Sigma$. Eq. 5.1 is applying marginalization to determine $\mathbf{x}_M$, where the weighting of each $\mathbf{x}_\Sigma$ is determined by g_i. The assignment $\mathbf{x}_a$ is assumed to be some known quantity; in our example $\mathbf{x}_a$ is the assignment to the command and

observation variables.

This problem can be equivalently described as sorting our $\mathbf{x}_M$ by the value of the summation of Eq. 5.1, in decreasing order, and then taking the first k elements. We refer to this as finding the k maximal $\mathbf{x}_M$ and use the operator $\arg^{\mathrm{k}}\max_{\mathbf{X}_M}$. In Eq. 5.1, $g_i\left(\mathbf{x}_M \cup \mathbf{x}_\Sigma \cup \mathbf{x}_a\right)$ is a problem-specific set of n functions, of which m appear in the numerator of the problem's objective function and $(n-m)$ appear in the denominator. We additionally allow each $\mathbf{x}_\Sigma$ to have an associated constant $\alpha_{\mathbf{x}_\Sigma}$. Eq. 5.1 finds assignments to $\mathbf{X}_M$ that maximize the objective, subject to $\mathbf{x}_a$.

If we let $\mathbf{x}_{M\Sigma a} = \mathbf{x}_M \cup \mathbf{x}_\Sigma \cup \mathbf{x}_a$, we specify that each $g_i\left(\mathbf{x}_{M\Sigma a}\right)$ has the form:

$$g_i\left(\mathbf{x}_{M\Sigma a}\right) = \sum_{\mathbf{x}'_R \in \mathbb{D}_{\mathbf{X}'_R \subseteq \mathbf{X}_R}} \prod_{j=0}^{n_i} f_j\left(\left(\mathbf{x}_{M\Sigma a} \Downarrow_{\mathbf{X}\setminus\mathbf{X}'_R}\right) \cup \mathbf{x}'_R\right) \mathcal{C}\left(\left(\mathbf{x}_{M\Sigma a} \Downarrow_{\mathbf{X}\setminus\mathbf{X}'_R}\right) \cup \mathbf{x}'_R\right) \tag{5.2}$$

where $\mathbf{X}'_R$ is an arbitrary subset of $\mathbf{X}_R$ and is specific to each g_i, with the restriction that every variable $\left(\mathbf{X}_R \setminus \mathbf{X}'_R\right) \subseteq \mathbf{X}_a$. That is to say that any variable we do not sum over must be a member of $\mathbf{X}_a$; this ensures that $\mathcal{C}$ can be evaluated. We use the notation $\mathbf{x}_{M\Sigma a} \Downarrow_{\mathbf{X}\setminus\mathbf{X}'_R}$ to denote the projection of the assignments $\mathbf{x}_{M\Sigma a}$ onto just the variables $\mathbf{X}\setminus\mathbf{X}'_R$. Let $\mathbf{x}' = \left(\mathbf{x}_{M\Sigma a} \Downarrow_{\mathbf{X}\setminus\mathbf{X}'_R}\right) \cup \mathbf{x}'_R$, that is, $\mathbf{x}'$ is an assignment to all $\mathbf{X}$ such that $\mathbf{x}'$ includes $\mathbf{x}'_R$ and for any variables not part of $\mathbf{x}'_R$, $\mathbf{x}'$ includes the corresponding assignments from $\mathbf{x}_{M\Sigma a}$. The function $\mathcal{C}\left(\mathbf{x}'\right)$ is 1 when $\mathbf{x}'$ is a solution to the CSP, and 0 otherwise. $f_j\left(\mathbf{x}'\right)$ specifies the value of the assignment $\mathbf{x}'$, and is problem specific. For example, in the estimation problem, f_j represents a transition probability. For some m^t and m^{t+1}, f_j returns $\mathbf{P}\left(m^{t+1}|m^t\right)$.

5.1 Overview of the Approach

In Eq. 5.2, $f_j\left(\mathbf{x}\right)$ is a VCSP constraint; it specifies a value of an assignment to multiple variables. We wish to leverage work on Optimal CSPs [59, 50], which require our value function to be a function of a unary variable. Thus, our approach to solving this OCSP is to first transform each f_j into an arity-one function using existing techniques from [50, 40]. These techniques work by creating a variable for each f_j with a domain element for every value in the image of the function. They

then add constraints that specify that each input to the function implies the appropriate assignment in the image of the function. The image of the function is then associated with the new f_j variable's assignments and thus the values are associated with a single variable. This encoding is presented in Section 5.2.

We then note that for all of our problems of interest, the numerator and denominator are often very similar in Eq. 5.1, and so through our reformulation of each g_i to an OCSP, we generalize the generated OCSP to a generic function g^*. To evaluate a specific g_i, we substitute appropriate values for each f_j and evaluate g^*.

We then proceed to compile

$$\arg^{\mathrm{k}}\max_{\mathbf{x}_M \in \mathbb{D}_{\mathbf{X}_M}} \sum_{\mathbf{x}_\Sigma \in \mathbb{D}_{\mathbf{X}_\Sigma}} \left(\prod_{i=1}^{m} g^*\left(\mathbf{x}_M \cup \mathbf{x}_\Sigma \cup \mathbf{x}_a, i\right) \cdot \prod_{i=m+1}^{n} \frac{1}{g^*\left(\mathbf{x}_M \cup \mathbf{x}_\Sigma \cup \mathbf{x}_a, i\right)} \cdot \alpha_{\mathbf{x}_\Sigma} \right) \tag{5.3}$$

where we need only compile one copy of g^*. g^* is an OCSP and can thus be compiled using existing techniques such as [19]. These existing compilation techniques find ways to re-order the sum and product terms so as to reduce the number of operations in the expression. The techniques achieve a compact representation through decomposition and the caching of identical sub-graphs as a single sub-graph. Our generalization uses the re-ordering of the sums and products to move products to the outer-most level of the $\arg^{\mathrm{k}}\max_{\mathbf{X}_M}$, which lets us use the commutativity of max and product to re-order the max and product terms. This reordering reduces the number of solutions over $\mathbf{x}_M$ that need to be considered. A review of the compilation techniques are given in Section 5.3.

We generalize the compilation of g^* to that of Eq. 5.3 by generalizing the compilation techniques of C2D [19]. Specifically, C2D compiles to an and-or graph called a Smooth Deterministic Decomposable Negation Normal Form (sd-DNNF) graph. We generalize the and-or graph to a graph that include the operator nodes:

- $\max^k$ – this operator captures the $\arg^{\mathrm{k}}\max$ operation, specifically selecting the k best solutions rooted at the node.

- $\max^k$ *combo* – this operator captures the $\times$ operation, but specific to the decomposition of the $\arg^{\mathrm{k}}\max$ operations. For a single solution $\max^k$ *combo* operates identical to $\times$, but for

larger numbers of solutions, i.e. $k > 1$, this operator selects the k best solutions rooted at the node. A solution of a $\max^k$ *combo* is the conjunction of a solution from each child. The value of a solution is the multiplication of the value of each child's solution.

- Iterator – This operator computes

$$\sum_{\mathbf{x}_\Sigma \in \mathbb{D}_{\mathbf{X}_\Sigma}} \left(\prod_{i=1}^{m} g^* \left(\mathbf{x}_M \cup \mathbf{x}_\Sigma \cup \mathbf{x}_a, i \right) \cdot \prod_{i=m+1}^{n} \frac{1}{g^* \left(\mathbf{x}_M \cup \mathbf{x}_\Sigma \cup \mathbf{x}_a, i \right)} \cdot \alpha_{\mathbf{x}_\Sigma} \right) \tag{5.4}$$

  for a particular $\mathbf{x}_M$ and has $g^* \left(\mathbf{x}_M \cup \mathbf{x}_\Sigma \cup \mathbf{x}_a, i \right)$ as its only child. Each g^* is compiled for a particular $\mathbf{x}_M$ and consists of only $+$ and $\times$ operators.

- $+$ – This is an ordinary addition operator. It combines the values of each of its children by adding them.

- $\times$ – This is an ordinary multiplication operator. It combines the values of each of its children by multiplying them.

where the sd-DNNF graphs normally support only two operations such as $+$ and $\times$. Each of these operators are placed in internal nodes to the graph. The graph is terminated by a set of literals. The literals are assignments x to some variable $X \in \mathbf{X}$. Each assignment appears only once in the graph.

The value of a literal depends on the context. For the variables created for each f_j, the literal evaluates to the appropriate value from the image of the function when evaluating the appropriate g_i and otherwise evaluates to the multiplicative identity, which is 1. For all other literals, they evaluate to 0 if the value is known to be false, for instance if another value is known to be the actual value of the variable, and is otherwise set to 1. Recall that g^* is a sum of products, so the multiplicative identity ensures that the value of a particular product is added to the total value of g^* as strictly the multiplication of each f_j. If, by contrast, one of the terms evaluates to 0 because it is false, then the value of 0 is added to summation and does not change the summation. This extension to the full set of operators is presented in Section 5.4.

We evaluate the compiled graph in two phases. In the first phase, we use the Iterator operator to

evaluate g^* once for each combination of each $\mathbf{x}_\Sigma$ and each of the n different g_i. For any particular $\mathbf{x}_\Sigma$ and g_i, all of the literals evaluate the same way, so significant amounts of sharing is possible between the multiple Iterator operators. This computation is thus done bottom-up for all the Iterator operators in parallel. Each Iterator then evaluates Eq. 5.4 on the resulting $n \cdot k$ values and produce a single value for itself. This algorithm is presented in Section 5.5.

For the evaluation of Eq. 5.1, these values are then used in the second phase to extract the k best solutions over $\mathbf{x}_M$ using these values. At this point the only operators left are the $\max^k$ and $\max^k$ *Combo* (MC) operators, along with the remaining literals $\mathbf{x}_M$. The algorithm works by annotating each node with the k best solutions rooted at that node based on its children's k best solutions. This is propagated up to the root node, at which point we have the k overall best solutions. We then extract the literals of each solution by following the annotations. This second phase problem is presented in Chapter 6.

An alternative problem that is also solved using the first phase is to sample a solution from the distribution encoded in the graph instead of extracting the k best solutions. This problem arises when we sample observations in order to compute the probability that the plan will succeed in the previous chapter. By sampling from the graph rather than from the k most likely $\mathbf{x}_M$, the samples will be from among all possible $\mathbf{x}_M$. The solution to this problem also has a lower time and space complexity than extracting the k best solutions. The solution to this problem replaces the $\max^k$ and $\max^k$ *Combo* operators in the graph with equivalent sampling operators, *Samp* and *Samp Combo*, respectively. These operators choose a random solution among their children based on the proportional value of the solutions rooted at each child. This choice is propagated from the root of the graph down to the literals, at which point the random solution is extracted. This algorithm is presented in Section 5.6.

5.2 Encoding the Value Function

In this section we introduce our encoding of f_j for compilation using the techniques of [50, 40]; we convert each f_j into a value function of a single variable as is required by our OCSP compilation technique. Specifically for each function $f_j(\mathbf{x})$, we add a variable to $\mathbf{X}$ that captures the values of

f_j, and we add constraints to $\mathcal{C}(\mathbf{x})$ to tie the assignments $\mathbf{x}$ to their value.

The idea of OCSP compilation techniques used here is that they naturally build on existing CSP compilation techniques. The hard constraints of the OCSP capture all inter-variable relationships, while the soft valuations are specific to individual variables. Thus, we can use a CSP compilation technique on the hard constraints and it is trivial to efficiently re-introduce the soft values for the individual variables in the compiled representation.

The technique in [50, 40] involves first creating a new variable X_j for the function f_j. For each unique value d in the image of f_j, we include a value x_d, indexed by d, in the domain of X_j. We then add these constraints to $\mathcal{C}(\mathbf{x})$:

$$x_d \Rightarrow \bigvee \{\mathbf{x} | f_j(\mathbf{x}) = d\} \tag{5.5}$$

For example, consider the valued relation corresponding to the transition relation of a fuel tank, from the estimation example in Chap. 7:

τ_{tank}

tank^t	flow^t	tank^{t+1}	$f_\tau(\mathbf{x})$
filled	zero	filled	1
filled	positive	filled	0.99
filled	positive	empty	0.01
empty	——	empty	1

(5.6)

Where the horizontal line indicates any value can be assigned to flow^t. We encode this by first adding a variable X_τ to our OCSP with the domain $\{x_1, x_{0.99}, x_{0.01}\}$. We then add the constraints:

$$\begin{aligned} x_1 &\Rightarrow \left(\text{filled}^t \wedge \text{zero}^t \wedge \text{filled}^{t+1}\right) \vee \left(\text{empty}^t \wedge \text{empty}^{t+1}\right) \\ x_{0.99} &\Rightarrow \text{filled}^t \wedge \text{positive}^t \wedge \text{filled}^{t+1} \\ x_{0.01} &\Rightarrow \text{filled}^t \wedge \text{positive}^t \wedge \text{empty}^{t+1} \end{aligned}$$

Where the idea is to multiply the value of a solution considered by g_i that includes x_1 by $f_\tau(x_1) = 1$ and multiply the value of a solution that includes $x_{0.99}$ by 0.99, etc.

We reformulate each f_j in this way, adding a variable and some constraints. We denote the new set of variables $\mathbf{X}_f$, one for each function f_j, and we use $\mathcal{C}'$ to denote $\mathcal{C}$ with the added new constraints. Using the reformulated functions, our evaluation of g_i from Eq. 5.2 is now:

$$g_i(\mathbf{x}_{M\Sigma a}) = \sum_{\{\mathbf{x}_f, \mathbf{x}_R\} \in \mathbb{D}_{\mathbf{X}_f, \mathbf{X}'_R}} \prod_{x \in \mathbf{x}_f} f_x(x)\, \mathcal{C}'\left(\left(\mathbf{x}_{M\Sigma a} \Downarrow_{\mathbf{X} \setminus \mathbf{X}'_R}\right) \cup \mathbf{x}'_R \cup \mathbf{x}_f\right) \tag{5.7}$$

Where the function f_x depends only on one variable, and they are combined with multiplication. What this specifically does for us is that it puts all inter-dependencies between variables within the characteristic function $\mathcal{C}'$. Thus, if we compile this characteristic function using a decomposition of its constraints, we can evaluate g_i directly on the compiled structure.

In order to evaluate g_i on the compiled structure, we rewrite Eq. 5.7 as a sum over the solutions consistent with our partial solution $\mathbf{x}$:

$$g_i(\mathbf{x}_{M\Sigma a}) = \sum_{\left\{\mathbf{x} | \mathbf{x} \in \mathbb{D}_{\mathbf{X}, \mathbf{X}_f} \wedge \mathcal{C}'(\mathbf{x}) = 1\right\}} \prod_{x \in \mathbf{x}_f} f_x(x) \prod_{x \in \mathbf{x} \Downarrow_{\mathrm{vars}(\mathbf{x}_{M\Sigma a}) \setminus \mathbf{X}'_R}} h_{\mathbf{x}_{M\Sigma a}}(x) \tag{5.8}$$

Where we have now introduced another new term $h_{\mathbf{x}_{M\Sigma a}}(x)$ that evaluates to 1 if $x \in \mathbf{x}_{M\Sigma a}$ and 0 otherwise, which is to say it evaluates to 1 if x is consistent with the value we have chosen for the variable of x in our parameter $\mathbf{x}_{M\Sigma a}$. This new term is also a function of just one variable. In this formulation, we are now summing over all solutions of $\mathcal{C}'$ and setting to zero the value of all solutions inconsistent with our premise $\mathbf{x}_{M\Sigma a}$.

If we then make one final extension by adding in a multiply by 1, the multiplicative identity:

$$g_i(\mathbf{x}_{M\Sigma a}) = \sum_{\left\{\mathbf{x} | \mathbf{x} \in \mathbb{D}_{\mathbf{X}, \mathbf{X}_f} \wedge \mathcal{C}'(\mathbf{x}) = 1\right\}} \prod_{x \in \mathbf{x}_f} f_x(x) \prod_{x \in \mathbf{x} \Downarrow_{\mathrm{vars}(\mathbf{x}_{M\Sigma a}) \setminus \mathbf{X}'_R}} h_{\mathbf{x}_{M\Sigma a}}(x) \prod_{x \in \mathbf{x}'_R} 1 \tag{5.9}$$

We now have an equation that still computes Eq. 5.2 but is now a sum over all the solutions of $\mathcal{C}'$, where each term evaluates to the product of a set of constants that depend only on one variable. This

is to say we have now specified a value for every assignment to a single variable. Since the structure of this formulation is independent of the input $\mathbf{x}_{M\Sigma a}$, except through a set of constants in $h_{\mathbf{x}_{M\Sigma a}}$, we can compile this equation in a way that is independent of $\mathbf{x}_{M\Sigma a}$. Given a compiled form of Eq. 5.9, we need only compute appropriate constants for each assignment to each individual variable, and then we can compute Eq. 5.9 directly on its compiled form.

We can clean up our equation if we let:

$$f_{\mathbf{x}_{M\Sigma a}}(x) = \begin{cases} f_x(x) & \text{var}(x) \in \mathbf{X}_f \\ 1 & \text{var}(x) \in \mathbf{X}'_R \\ h_{\mathbf{x}_{M\Sigma a}}(x) & \text{Otherwise} \end{cases} \tag{5.10}$$

$$g_i(\mathbf{x}_{M\Sigma a}) = \sum_{\left\{\mathbf{x} | \mathbf{x} \in \mathbb{D}_{\mathbf{X}, \mathbf{X}_f} \wedge \mathcal{C}'(\mathbf{x}) = 1\right\}} \prod_{x \in \mathbf{x}} f_{\mathbf{x}_{M\Sigma a}}(x) \tag{5.11}$$

In this form, it is apparent that we can support changing the values of any of the individual assignments x of $f_{\mathbf{x}_{M\Sigma a}}(x)$. In general, there is no need to change the values of $f_x(x)$. An important consequence of our ability to change the values of $h_{\mathbf{x}_{M\Sigma a}}$ is that this formulation can support any choice of $\mathbf{X}'_R \subseteq \mathbf{X}_R$, so long as $\mathbf{x}_{M\Sigma a}$ specifies values for any $\mathbf{X}_R$ not included in $\mathbf{X}'_R$, and hence $h_{\mathbf{x}_{M\Sigma a}}$ is well defined. We can support this choice of $\mathbf{X}'_R$ by modifying $h_{\mathbf{x}_{M\Sigma a}}$. If we compile g_i initially for $\mathbf{X}'_R = \{\}$, then we can support any $\mathbf{X}'_R$ by modifying $h_{\mathbf{x}_{M\Sigma a}}(x)$ to evaluate to 1 for any $\text{var}(x) \in \mathbf{X}'_R$.

The next section shows existing approaches for compiling $\mathcal{C}'$ in a way suitable for computing Eq. 5.11 over all the solutions of $\mathcal{C}'$.

5.3 Compilation Techniques

We present below existing techniques to compile the solutions of the constraints $\mathcal{C}'$, and how these techniques allow for the evaluation of the sum of products given in Eq. 5.11. Specifically, if we compile $\mathcal{C}'$ such that the encoding contains each solution to $\mathcal{C}'$ once, which is to say, all $\mathbf{x}$ that evaluate to 1, then we can evaluate Eq. 5.11 by enumerating the solutions in the encoding and for each solution accumulating the result of evaluating our function $f_{\mathbf{x}_{M\Sigma a}}$. In our approach, we avoid the brute-force enumeration by taking advantage of the compiled form; any compact compiled form embodies two ideas: first, selecting between possible partial solutions and second, concatenating partial solutions. For the evaluation of Eq. 5.11, the former corresponds to addition, the latter corresponds to multiplication by the value of each partial solution, and the value of a single assignment in a partial solution is $f_{\mathbf{x}_{M\Sigma a}}$. We use this correspondence to compute Eq. 5.11 on the compiled form by identifying the operation for selecting and for concatenating.

For the OCSP problem, the $\max^k$ and $+$ operators are selection operators, and the $\max^k$ *combo* and $\times$ operators are concatenation operators. We additionally have a special operator Iterator that separates our maximization operators from our sum-of-products operators. Since our two pairs of operators are separated by a special operator, we can embed both pairs of operators in the same representation that normally only supports one pair of operators so long as the representation respects the separation of these two pairs of operators. Specifically, using the compiled representation reviewed next, we can compile the OCSP into a graph representation that has the $\max^k$ operators in the top half of the graph and the $+$ and $\times$ operators in the bottom half. These two halves preserve the Iterator operators, which are otherwise not embedable in the representation. Using this approach, we can leverage existing algorithms and compilers for this compiled representation. Section 5.4 shows how to extend the compiled representation so it respects the encoding of the OCSP, Eq. 5.1, using all five operators.

5.3.1 Compiled Constraint Representation

We start with an explicit and-or graph formulation called a Smooth Deterministic Decomposable Negation Normal Form (sd-DNNF) [18]; an example sd-DNNF is pictured in Fig. 5-1. The sd-

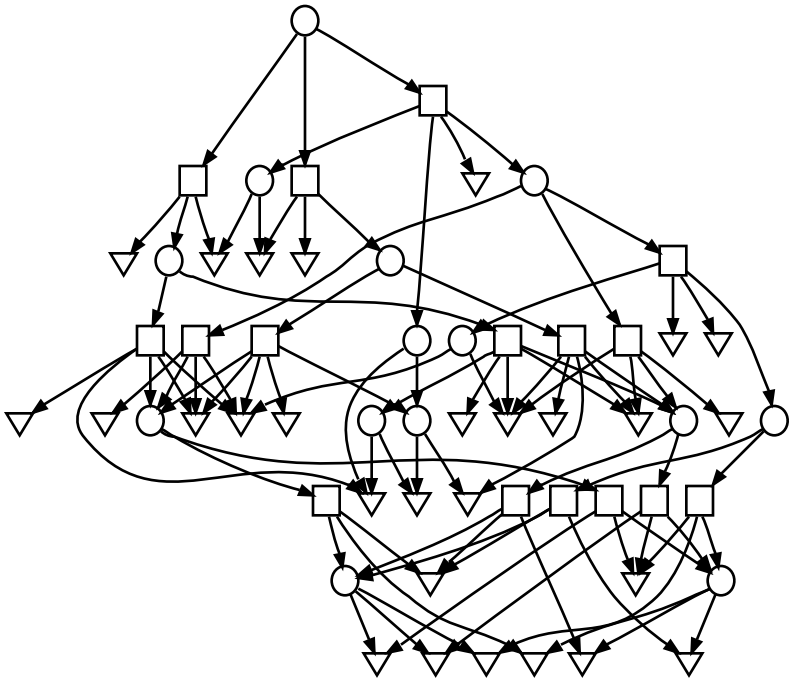

Figure 5-1: This is an example sd-DNNF, a compact solution representation. This example contains 180 solutions within the 58 nodes, where a selection typically includes about 20 nodes. We use circles to represent **Or** nodes, squares for **And** nodes, and triangles for leaf nodes.

DNNF representation is a directed acyclic graph, with a single root node. The internal nodes are either **And** or **Or** nodes, and the leaves of the graph are each labeled with an assignment to one of the variables. There is a compiler called C2D [19] that takes as input a constraint in Conjunctive Normal Form (CNF) and generates an sd-DNNF representation for that constraint.

An sd-DNNF compactly encodes each solution to a constraint exactly once within the graph. Since our OCSP equation is specified as a computation on the solutions to the constraint of the OCSP, it naturally follows to embed the computation within the compact representation. For example, if we compile g_i, then the **And** nodes are multiplication operators, the **Or** nodes are addition operators, and the leaves are labeled with values from $f_{\mathbf{x}_{M\Sigma a}}$. With values and operators on the sd-DNNF, the sd-DNNF is called a *valued sd-DNNF*, which has been used to encode the Minimum Cardinality (MCard) problem in [18] and to encode sums of products expressions in [2]. To com-

pute g_i using the sd-DNNF, we use the values f_x, $h_{\mathbf{x}}$, and 1 on the leaves, as defined by $f_{\mathbf{x}_{M\Sigma a}}$, and apply the operators in the graph, reading off the answer at the root.

We next review the valued sd-DNNF representation and the properties that make it a compact representation for our problem.

5.3.2 Valued sd-DNNF

We review the properties of a valued sd-DNNF in this section. As shown in Fig. 5-1, an sd-DNNF is an acyclic graph of **And** and **Or** nodes, terminated by **Leaf** nodes. A valued sd-DNNF introduces a function that labels each **Leaf** node with a value and a pair of operators for combining these values at the **And** and **Or** nodes. For example, suppose we use $\times$ and $\max$ for the **And** and **Or** nodes, respectively, and real numbers for the values at the leaves for a maximization problem. The value at the root of the graph after applying each $\times$ and $\max$ operator corresponds to the product of the values of the leaves that result in the highest-valued solution.

We define a valued sd-DNNF by the tuple $\langle V, E, \mathcal{L}_L, \mathcal{L}_{\mathbf{P}}, \times, + \rangle$:

- V is the set of nodes of the directed, acyclic graph. V is partitioned into three sets, A, O, and L, corresponding to **And**, **Or**, and **Leaf** nodes, respectively. $r \in V$ is the single root node of the graph.

- E is the set of edges of the graph. An edge $e \in E$ is an ordered pair of nodes $\langle m, n \rangle$. Edges are directed from m to n. Leaves have no out-going edges; hence, $m \in A \cup O$ and $n \in V = A \cup O \cup L$. We define a *path* $v_1, v_2, \ldots, v_p$ in the standard way: for every successive pair of nodes v_i and v_{i+1} in the path, there is an edge $\langle v_i, v_{i+1} \rangle \in E$. The graph is acyclic, hence no path exists such that $v_1 = v_p$, for any $p > 1$. For each edge $\langle m, n \rangle$, we designate n as a *child* of m, and m as a *parent* of n. All **And** and **Or** nodes have at least one child.

- $\mathcal{L}_L$ is a function that labels each element of L with a unique symbol or the empty symbol $\emptyset$. To compute g_i, this unique symbol is an assignment x to some variable X and never the empty symbol. $\emptyset$ is used to introduce maximization later, specifically the empty symbol allows us to introduce the values computed by g_i into the maximization equation, where the values do not

have associated assignments.

- $\mathcal{L}_{\mathbf{P}}$ is a function that labels the elements of L with a value. For our OCSP, the values are one of 0, 1, and the image of each f_j.
- $\times : \text{values} \times \text{values} \rightarrow \text{values}$ is a binary function used to compose $\mathcal{L}_{\mathbf{P}}$ values at **And** nodes.
- $+$ is a binary function used to combine $\mathcal{L}_{\mathbf{P}}$ values at **Or** nodes.

Note that $\times$ and $+$ are the operators appropriate for computing sums of products. In general, the operator appropriate for **And** nodes is the one appropriate for concatenating partial solutions and the operator appropriate for **Or** nodes is the one appropriate for combining the values among multiple solutions. Later we use $\max^k$ instead of $+$ and $\max^k$ *Combo* instead of $\times$, when we perform maximization.

Selection

A solution to the constraint encoded in the sd-DNNF is extracted from the sd-DNNF by selecting the appropriate children at the nodes of the graph. For an sd-DNNF, a selection implies a particular solution, and the selection is the set of nodes that define the solution, while the solution is the set of symbols on the leaves of the selection. More precisely, a *selection* is a set of nodes that obey these rules:

1. A selection always includes the root node.
2. For every **And** node a selected, every child of a is also selected.
3. For every **Or** node o selected, one and only one child of o is also selected.

For example, consider the selection of Fig. 5-2. The selection shown is $\{$o1, a2, l4$\}$. We denote the root o1 with a double line. In this case, our root is an **Or** node. The other two valid selections are $\{$o1, o3, a5, l7$\}$ and $\{$o1, o3, a6, l8$\}$. The labels of $\mathcal{L}_L$ and of $\mathcal{L}_{\mathbf{P}}$ are designated in the figure within the $L[\ldots]$ and the $P[\ldots]$ leaf labels, respectively, of nodes l4, l7, and l8. For example, $\mathcal{L}_L$ (l4) $=$ "Switch=Off" and $\mathcal{L}_{\mathbf{P}}$ (l4) $= 0.5$. A solution, as is shown in Section 5.3.2, is the set of

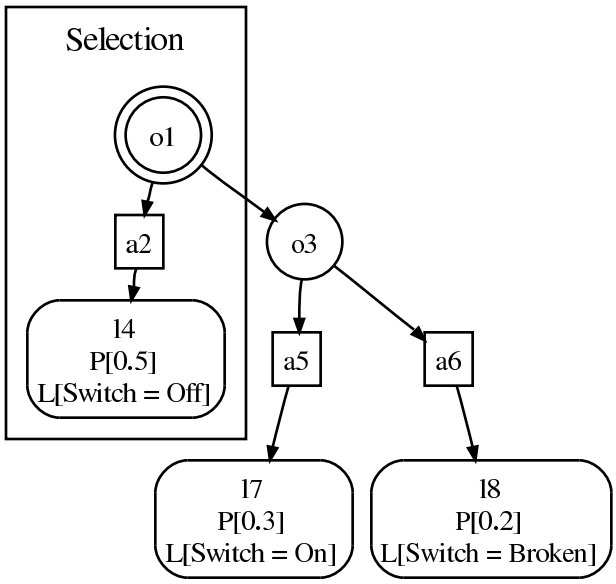

Figure 5-2: This figure shows a selection of this simple sd-DNNF. The node o1 is the root of the tree. Our selection consists of o1, a2, and l4. This is a correct selection as it includes the root o1; it includes exactly one child of o1, namely a2; and it includes all of the children of a2, namely l4.

leaf symbols of a selection. We now explain the properties of an sd-DNNF that make this statement true.

sd-DNNF Properties

An sd-DNNF[18] imposes three properties on an and-or acyclic graph: *smooth*, *deterministic*, and *decomposable*. All three properties assume that the symbols we use to label the leaves represent assignments to variables, either binary or multi-valued.

The *smooth* property states that every variable X that labels a leaf descendant of one child of an **Or** node must label a leaf descendant of all children of the **Or** node. Said another way, for an **Or** node o, every selection rooted at o defines an assignment containing exactly the same variables as every other selection rooted at o. Fig. 5-2 is smooth because the only variable, Switch, appears on a leaf of some descendant of both children of both **Or** nodes, o1 and o3. The smooth property

ensures that, when computing g_i, the sd-DNNF representation can be evaluated by applying the $+$ operator as we have defined it. Without smoothness, some sub-graphs can omit some variables if any value is consistent, while the value of that sub-graph must be computed using all the different values. Thus, without smoothness, it is necessary to keep track of which variables are not assigned values under each child of the $+$ operator and explicitly sum all the values of that variable for that branch.

The *deterministic* property applies to **Or** nodes. This property requires that each selection rooted at the **Or** node must denote a different set of assignments to the variables. For Fig. 5-2, the deterministic property trivially holds, as each selection has a different leaf and each leaf has a unique assignment. The deterministic property ensures that solutions are not double counted in the summation of g_i.

The *decomposable* property applies to **And** nodes. This property requires that the variables of the leaves of a descendant of a child of the **And** node are disjoint from the variables of any other descendant of every other child; an **And** node partitions variables among its children. In this way, contradictory assignments are never included in the same selection. The decomposable property ensures that inconsistent solutions that include multiple assignments to the same variable are not included in the summation of g_i.

The *decomposable* property ensures that a selection in the graph is a tree and not a graph. If a selection were a graph, then there would be some node a from which we could take two different edges out of the node and end up at the same node r by these two different paths. By the definition of a selection, a must be an **And** node as all other nodes in a selection have less than two out-going edges. Since we can reach r from two edges of the **And** node, this **And** node has an assignment to the same variable(s) among at least two children, namely all the assignments rooted at r, which violates the *decomposable* property.

Solution

A solution of a valued sd-DNNF denotes a complete set of assignments to the variables. A *solution* is constructed by creating a *selection* of nodes, and then applying $\mathcal{L}_L$ to all of the leaf nodes of the

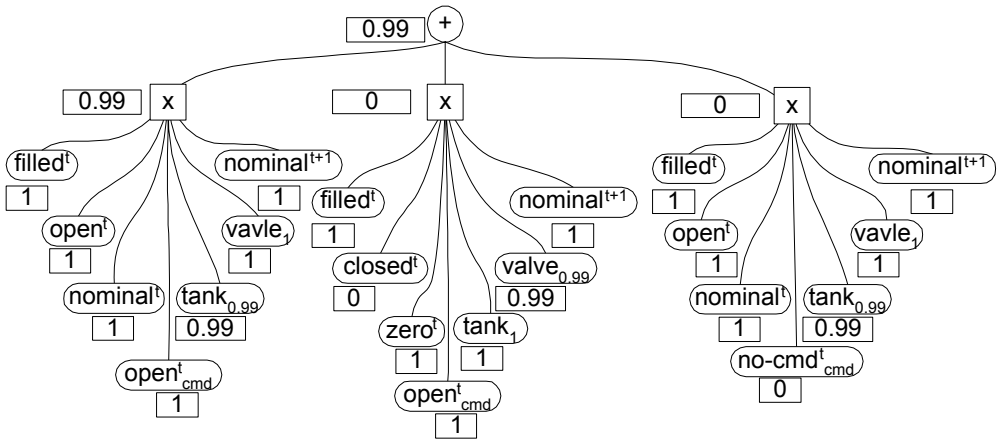

Figure 5-3: This tree shows a way to compute g_i using Eq. 5.11 for the example given in Table 5.2. This particular figure shows the computation for $\mathbf{x}_\Sigma = \{\text{filled}^t, \text{open}^t\}$ and $\mathbf{x}_a = \{\text{open}^t_{\text{cmd}}\}$. This valued and-or tree employs no compilation techniques.

selection. The set of resulting leaf symbols is a solution. We omit the empty symbol $\emptyset$ from our solution, should it exist.

For example, the solution of the selection $\{$o1, a2, l4$\}$ shown in Fig. 5-2 is $\{$"Switch=Off"$\}$. Since the only leaf node of this selection is l4, our solution is the set containing only $\mathcal{L}_L$ (l4). As stated above, this is "Switch=Off".

We assume our and-or graph is deterministic, hence each selection denotes a unique set of leaf symbols. We further require that these set of leaf symbols are unique even after omitting the empty symbol $\emptyset$, and thus each selection has a unique solution.

Solution's Value

To compute the *value* of a solution, we apply $\mathcal{L}_\mathbf{P}$ to all of the leaf nodes of the unique selection[1] of the solution and then combine them with $\times$. In the example of Fig. 5-2, the value of the selection and solution is 0.5. Since we only have one leaf, we need not apply $\times$.

[1] We can assume that selections and solutions are uniquely paired because we assume that there is at most one selection in an sd-DNNF that generates a particular solution. The deterministic property of an sd-DNNF ensures this is a correct assumption.

tank^t	valve^t	vp^t_{out}	$\text{valve}^t_{\text{cmd}}$	tank^{t+1}	valve^{t+1}	$\text{vp}^{t+1}_{\text{out}}$	τ_{tank}	τ_{valve}
filled	open	nominal	open	filled	open	nominal	0.99	1
filled	closed	zero	open	filled	open	nominal	1	0.99
filled	open	nominal	no-cmd	filled	open	nominal	0.99	1
filled	open	nominal	close	filled	closed	zero	0.99	1
filled	closed	zero	close	filled	closed	zero	1	0.99
filled	closed	zero	no-cmd	filled	closed	zero	1	0.99
filled	closed	zero	open	filled	stuck	zero	1	0.01
filled	stuck	zero	open	filled	stuck	zero	1	1
filled	closed	zero	close	filled	stuck	zero	1	0.01
filled	stuck	zero	close	filled	stuck	zero	1	1
filled	closed	zero	no-cmd	filled	stuck	zero	1	0.01
filled	stuck	zero	no-cmd	filled	stuck	zero	1	1
filled	open	nominal	open	empty	open	zero	0.01	1
empty	open	zero	open	empty	open	zero	1	1
empty	closed	zero	open	empty	open	zero	1	0.99
filled	open	nominal	no-cmd	empty	open	zero	0.01	1
empty	open	zero	no-cmd	empty	open	zero	1	1
filled	open	nominal	close	empty	closed	zero	0.01	1
empty	open	zero	close	empty	closed	zero	1	1
empty	closed	zero	close	empty	closed	zero	1	0.99
empty	closed	zero	no-cmd	empty	closed	zero	1	0.99
empty	closed	zero	open	empty	stuck	zero	1	0.01
empty	stuck	zero	open	empty	stuck	zero	1	1
empty	closed	zero	close	empty	stuck	zero	1	0.01
empty	stuck	zero	close	empty	stuck	zero	1	1
empty	closed	zero	no-cmd	empty	stuck	zero	1	0.01
empty	stuck	zero	no-cmd	empty	stuck	zero	1	1

Table 5.1: A table of consistent solutions for the mono-propellant propulsion system, shown in Fig. 7-1. This is an exhaustive list.

tank^t	valve^t	vp^t_{out}	$\text{valve}^t_{\text{cmd}}$	$\text{vp}^{t+1}_{\text{out}}$	τ_{tank}	τ_{valve}
filled	open	nominal	open	nominal	0.99	1
filled	closed	zero	open	nominal	1	0.99
filled	open	nominal	no-cmd	nominal	0.99	1

Table 5.2: A table of solutions for the mono-propellant propulsion system consistent with $\mathbf{x}_M = \left\{\text{filled}^{t+1}, \text{open}^{t+1}\right\}$.

5.3.3 sd-DNNF Compilation

This representation uses two techniques to compactly encode the and-or graph: memoization and decomposition [20, 19]. Both techniques are illustrated on a fuel tank and mono-propellent thruster example, as illustrated in Fig. 7-1. The solutions to this example's constraints are shown in Table 5.1. In this example we consider the OCSP g_i that has two functions f_i, which are converted to variables using the techniques of Section 5.2. These variables are labeled τ_{tank} and τ_{valve}. For this example, $\mathbf{X}'_R = \left\{\text{vp}^t_{\text{out}}, \text{vp}^{t+1}_{\text{out}}\right\}$ and $\mathbf{x}_M = \left\{\text{filled}^{t+1}, \text{open}^{t+1}\right\}$. For this example, we simplify Table 5.1 to Table 5.2 by restricting our tuples to those consistent with $\mathbf{x}_M$.

The valued and-or graph encoding of Table 5.2 is given in Fig. 5-3. For the example, we are computing:

$$g_1\left(\mathbf{x}_{M\Sigma a}\right) = \sum_{\mathbf{x}\in\text{Table 5.1}} \prod_{x\in\mathbf{x}} f_{\mathbf{x}_{M\Sigma a}}\left(x\right) \tag{5.12}$$

Where

$$\begin{aligned}
\mathbf{x}_M &= \left\{\text{filled}^{t+1}, \text{open}^{t+1}\right\} & \mathbf{X}_f &= \left\{\tau_{\text{tank}}, \tau_{\text{valve}}\right\} \\
\mathbf{x}_\Sigma &= \left\{\text{filled}^{t}, \text{open}^{t}\right\} & \mathbf{x}_a &= \left\{\text{open}^t_{\text{cmd}}\right\} \\
\mathbf{X}'_R &= \left\{\text{vp}^t_{\text{out}}, \text{vp}^{t+1}_{\text{out}}\right\} & \mathbf{x}_{M\Sigma a} &= \mathbf{x}_a \cup \mathbf{x}_M \cup \mathbf{x}_\Sigma \\
f_{\mathbf{x}_{M\Sigma a}}\left(x\right) &= \begin{cases} f_x\left(x\right) & \text{var}\left(x\right) \in \left\{\tau_{\text{tank}}, \tau_{\text{valve}}\right\} \\ 1 & \text{var}\left(x\right) \in \left\{\text{vp}^t_{\text{out}}, \text{vp}^{t+1}_{\text{out}}\right\} \\ h_{\mathbf{x}_{M\Sigma a}}\left(x\right) & \text{Otherwise} \end{cases} \\
f_x\left(x\right) &= \begin{cases} 0.99 & x \in \left\{\text{tank}_{0.99}, \text{valve}_{0.99}\right\} \\ 1 & x \in \left\{\text{tank}_1, \text{valve}_1\right\} \end{cases} \\
h_{\mathbf{x}_{M\Sigma a}}\left(x\right) &= \begin{cases} 1 & x \in \left\{\text{filled}^t, \text{open}^t, \text{filled}^{t+1}, \text{open}^{t+1}\right\} \\ 0 & \text{Otherwise} \end{cases}
\end{aligned}$$

Evaluating the multiplication of the left sub-graph in Fig. 5-3 on page 83 corresponds to the first row of Table 5.2. Evaluating the middle cluster corresponds to the second row and the right cluster corresponds to the third and final row. To illustrate this correspondence with Fig. 5-3, we expand

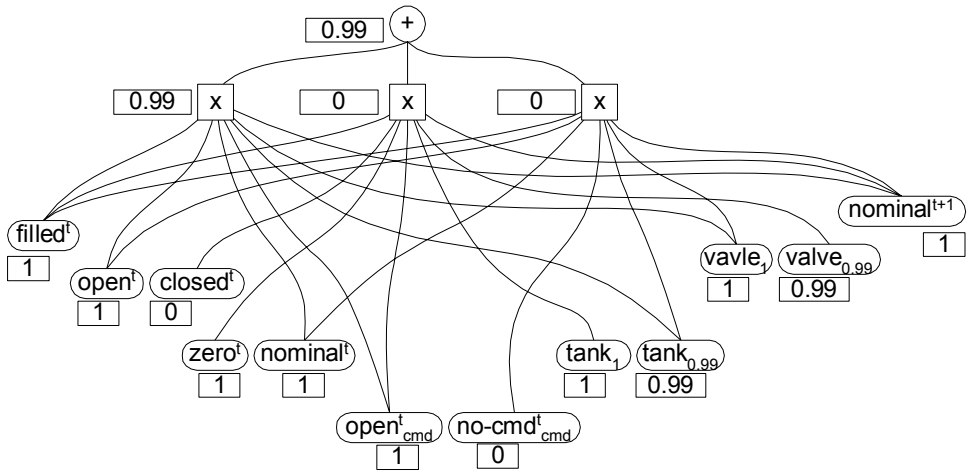

Figure 5-4: This graph shows how to compute the same equation g_1 over Table 5.2 as is computed in Fig. 5-3. Here we have combined terms that are identical in Fig. 5-3.

Eq. 5.12:

$$
\begin{aligned}
g_1\left(\mathbf{x}_{M\Sigma a}\right) = \Big[& f_{\mathbf{x}_{M\Sigma a}}\left(\text{filled}^t\right) f_{\mathbf{x}_{M\Sigma a}}\left(\text{open}^t\right) f_{\mathbf{x}_{M\Sigma a}}\left(\text{nominal}^t\right) f_{\mathbf{x}_{M\Sigma a}}\left(\text{open}^t_{\text{cmd}}\right) \cdot \\
& f_{\mathbf{x}_{M\Sigma a}}\left(\text{nominal}^{t+1}\right) f_{\mathbf{x}_{M\Sigma a}}\left(\text{tank}_{0.99}\right) f_{\mathbf{x}_{M\Sigma a}}\left(\text{valve}_1\right) \Big] \\
+ \Big[& f_{\mathbf{x}_{M\Sigma a}}\left(\text{filled}^t\right) f_{\mathbf{x}_{M\Sigma a}}\left(\text{closed}^t\right) f_{\mathbf{x}_{M\Sigma a}}\left(\text{zero}^t\right) f_{\mathbf{x}_{M\Sigma a}}\left(\text{open}^t_{\text{cmd}}\right) \cdot \\
& f_{\mathbf{x}_{M\Sigma a}}\left(\text{nominal}^{t+1}\right) f_{\mathbf{x}_{M\Sigma a}}\left(\text{tank}_1\right) f_{\mathbf{x}_{M\Sigma a}}\left(\text{valve}_{0.99}\right) \Big] \\
+ \Big[& f_{\mathbf{x}_{M\Sigma a}}\left(\text{filled}^t\right) f_{\mathbf{x}_{M\Sigma a}}\left(\text{open}^t\right) f_{\mathbf{x}_{M\Sigma a}}\left(\text{nominal}^t\right) f_{\mathbf{x}_{M\Sigma a}}\left(\text{no-cmd}^t_{\text{cmd}}\right) \cdot \\
& f_{\mathbf{x}_{M\Sigma a}}\left(\text{nominal}^{t+1}\right) f_{\mathbf{x}_{M\Sigma a}}\left(\text{tank}_{0.99}\right) f_{\mathbf{x}_{M\Sigma a}}\left(\text{valve}_1\right) \Big]
\end{aligned}
\tag{5.13}
$$

Memoization

We can reduce the number of edges and nodes of an and-or tree by applying the first compilation technique called memoization. Memoizing involves caching a single copy of identical subexpressions and sharing the copy among all expressions. Applying memoization to an and-or tree creates a more compact and-or graph.

For example, if we apply memoization to Fig. 5-3, we get Fig. 5-4. Due to the simplicity of this

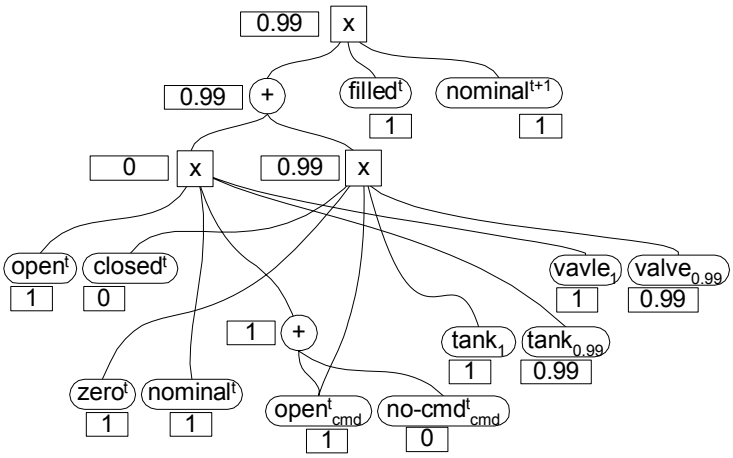

Figure 5-5: This graph shows how to compute the same equation g_1 over Table 5.2 as Fig. 5-4 with fewer operations. We have combined sub-expressions and interleaved our addition and multiplication to reduce the number of computations.

example, we only reduce the number of nodes: from 25 to 16. We do not eliminate any of the 24 edges. Both graphs still require 18 multiplications and 2 additions. Memoization does not visibly alter Eq. 5.13 as the lack of repeated sub-expressions is hard to represent in a textual equation.

Decomposition

The second compilation technique is decomposition. Decomposition looks for two or more disjoint sets of variables, call them $\mathbf{A}$ and $\mathbf{B}$, that are conditionally independent with respect to the solutions to the constraints, given an assignment $\mathbf{c}$ to a third disjoint set of variables $\mathbf{C}$. Decomposition means that the solutions to all constraints that include $\mathbf{c}$ are a cross product of these solutions projected onto the $\mathbf{A}$ and $\mathbf{B}$ variables. Said in another way, for any $\mathbf{a}$ and $\mathbf{b}$ such that $\mathbf{a} \cup \mathbf{c}$ is a subset of a solution to the constraints and $\mathbf{b} \cup \mathbf{c}$ is a subset of a solution to the constraints, then $\mathbf{A}$ and $\mathbf{B}$ are conditionally independent given $\mathbf{c}$ if and only if $\mathbf{a} \cup \mathbf{b} \cup \mathbf{c}$ is a subset of a solution to the constraints.

For the and-or graph in this thesis, this conditional independence property allows us to factor a

sum of products into a product of two sums:

$$\left(\sum_{\mathbf{a}\in\mathbb{D}_{\mathbf{A}}}\left(\sum_{\mathbf{b}\in\mathbb{D}_{\mathbf{B}}} f\left(\mathbf{a},\mathbf{b},\mathbf{c}\right)\right)\right) = \left(\sum_{\mathbf{a}\in\mathbb{D}_{\mathbf{A}}}\left(\sum_{\mathbf{b}\in\mathbb{D}_{\mathbf{B}}} f_1\left(\mathbf{a},\mathbf{c}\right)\cdot f_2\left(\mathbf{b},\mathbf{c}\right)\right)\right)$$
$$= \left(\sum_{\mathbf{a}\in\mathbb{D}_{\mathbf{A}}} f_1\left(\mathbf{a},\mathbf{c}\right)\right)\cdot\left(\sum_{\mathbf{b}\in\mathbb{D}_{\mathbf{B}}} f_2\left(\mathbf{b},\mathbf{c}\right)\right)$$

For example, applying decomposition to Eq. 5.13 yields:

$$\begin{aligned} g_1\left(\mathbf{x}_{M\Sigma a}\right) &= f_{\mathbf{x}_{M\Sigma a}}\left(\text{filled}^t\right) f_{\mathbf{x}_{M\Sigma a}}\left(\text{nominal}^{t+1}\right)\cdot \\ &\quad\Bigg[\Bigg[f_{\mathbf{x}_{M\Sigma a}}\left(\text{open}^t\right) f_{\mathbf{x}_{M\Sigma a}}\left(\text{nominal}^t\right) f_{\mathbf{x}_{M\Sigma a}}\left(\text{tank}_{0.99}\right) f_{\mathbf{x}_{M\Sigma a}}\left(\text{valve}_1\right)\cdot \\ &\qquad\qquad\left[f_{\mathbf{x}_{M\Sigma a}}\left(\text{open}^t_{\text{cmd}}\right) + f_{\mathbf{x}_{M\Sigma a}}\left(\text{no-cmd}^t_{\text{cmd}}\right)\right]\Bigg] \\ &\quad+\Bigg[f_{\mathbf{x}_{M\Sigma a}}\left(\text{closed}^t\right) f_{\mathbf{x}_{M\Sigma a}}\left(\text{zero}^t\right) f_{\mathbf{x}_{M\Sigma a}}\left(\text{open}^t_{\text{cmd}}\right)\cdot \\ &\qquad\qquad f_{\mathbf{x}_{M\Sigma a}}\left(\text{tank}_1\right) f_{\mathbf{x}_{M\Sigma a}}\left(\text{valve}_{0.99}\right)\Bigg]\Bigg] \end{aligned} \tag{5.14}$$

For the graph shown in Fig. 5-5, this corresponds to re-ordering the sum and product nodes in the same manner as in the equation. The resulting graph is shown in Fig. 5-5. Both graphs denote the same expression, Eq. 5.13. Applying decomposition and rearranging the graph increases the number of nodes from 16 to 17, but reduces the number of edges from 24 to 17 and the number of operations from 18 multiplications and 2 additions to 10 multiplications and 2 additions. This is a net savings of 8 operations for a simple example.

There are only trivial instances of conditional independence in this example, as one of the two variable sets $\mathbf{A}$ and $\mathbf{B}$ end up only having one solution. For example $\mathbf{A} = \left\{\text{valve}^t_{\text{cmd}}\right\}$ and $\mathbf{B} = \left\{\text{vp}^t_{\text{out}}\right\}$ are conditionally independent given $\mathbf{c} = \left\{\text{open}^t, \text{filled}^t\right\}$, but since $\left(\text{open}^t, \text{filled}^t\right) \Rightarrow \text{nominal}^t$, $\mathbf{B}$ only has one solution; in this example $\mathbf{A}$ has two solutions.

Re-examining the full set of solutions in Table 5.1, a non-trivial example of conditional independence is $\mathbf{A} = \left\{\text{tank}^t, \text{tank}^{t+1}, \tau_{\text{tank}}\right\}$, $\mathbf{B} = \left\{\text{valve}^t_{\text{cmd}}\right\}$, and $\mathbf{c} = \left\{\text{stuck}^t\right\}$. In this example, $\mathbf{A}$ has two partial solutions, which specify that the tank stays full or stays empty and with $\mathbf{B}$, the

command can take on any of its three values. Table 5.1 contains all six combinations.

The C2D [19] package provides both Memoization and Decomposition algorithms that we utilize in this thesis. The next section expands on conditional independence introduced here, in order to extend decomposition to our entire OCSP, Eq. 5.1.

5.4 Extending to Maximization

Using the techniques presented in the previous section, we are capable of compiling $g_i\left(\mathbf{x}_{M\Sigma a}\right)$ in a way that is independent of the particular input assignment $\mathbf{x}_{M\Sigma a}$, thus we can evaluate this compiled form for any assignment $\mathbf{x}_{M\Sigma a}$. This compiled form is also independent of our choice of the variables $\mathbf{X}'_R$, as long as $\mathbf{x}_{M\Sigma a}$ specifies values for any $\mathbf{X}_R$ not included in $\mathbf{X}'_R$. In order to evaluate our OCSP Eq. 5.1 using this compiled representation, one must evaluate $g_i\left(\mathbf{x}_{M\Sigma a}\right)$ for every $\mathbf{x}_M$ and $\mathbf{x}_\Sigma$ in the equation. For our OCSP, we assume there are only k assignments $\mathbf{x}_\Sigma$, as we are recursively solving this problem, but there are in general exponentially many assignments $\mathbf{x}_M$. We thus introduce decomposition into our maximization calculation to reduce the number of $\mathbf{x}_M$ that need to be evaluated.

Our insight is that maximization of two (conditionally) independent multiplications commutes as the multiplication of two (conditionally) independent maximizations. Based on this insight, decomposition allows us to identify conditional independence at this outer most level of the OCSP problem and thus break it down into smaller pieces.

In order to understand how to correctly compile Eq. 5.1, let us return to our earlier point that the compiled form has two notions: first, selecting between possible partial solutions towards the goal of generating a complete solution, and second, joining partial solutions. Computing $\max^k$ corresponds to selecting the best value among possible partial solutions, which corresponds to the first notion. The $\max^k$ *Combo* operator corresponds to the second notion.

We wish to embed both the $+$ and $\times$ operators, corresponding to computing g^*, and the $\max^k$ and $\max^k$ *Combo* operators in the and-or graph to maximize our memoization and decomposition ability, while still maintaining correctness. We are using a tool and representation that only preserves two operators - one for each of the above notions. In order to take advantage of this tool, we

need to ensure that the sd-DNNF compiled by the tool maintains the distinction between the $\max^k$ and arithmetic operators despite it not otherwise being aware of such a difference. Additionally, it must be possible to re-identify the locations of the Iterator operators.

We will show momentarily that the $\max^k$ and arithmetic operators are naturally separated by the Iterator operators. Since the maximization sub-problem is deciding between different assignments $\mathbf{x}_M$ in the top part of the graph of the equation, we need to ensure that the sd-DNNF also decides between different assignments $\mathbf{x}_M$ in the top part of the compiled graph. When we embed the OCSP problem into the sd-DNNF, we can identify the $\max^k$ operations because they are deciding between assignments $\mathbf{x}_M$. The Iterator operators are inserted directly below these identified $\max^k$ operators when recovering our and/or graph representation from the compiled sd-DNNF.

Given our discussion about conditional independence above, we can additionally include those variables $\mathbf{X}_a$ that always have values in the top of our tree as exactly one of the assignments to these variables is 1 and the rest are 0. When these are included in the maximization, the inconsistent assignments zero the value of all solutions that included the inconsistent assignment, eliminating it from the maximization. The additional flexibility of including these variables in either half of the graph may lead to a more compact representation.

Recall the definition of the $\max^k$ operator. If we assume that a child of a $\max^k$ node has up to k solutions, this operator is choosing the k best solutions from among its children's solutions. We use a $\max^k$ node when choosing between solutions to the variables $\mathbf{X}_M$ and use $+$ when choosing between solutions to the remaining variables.

Recall the definition of the $\max^k$ *Combo* operator. If we only sought one assignment, then simple multiplication correctly captures how to combine the value of this operator. To find k assignments, a $\max^k$ *Combo* operator must consider combinations of its children's k solutions to find its own k best combinations. We abbreviate $\max^k$ *Combo* as MC.

We have thus far been compiling g_i for particular $\mathbf{x}_M$ in preparation for adding in these additional operators. Armed with these additional operators, we can now compile the full set of solutions, Table 5.1, into our extended sd-DNNF, as shown in Fig. 5-6.

Using the smooth and deterministic properties of sd-DNNFs, we can prove that $\max^k$ nodes

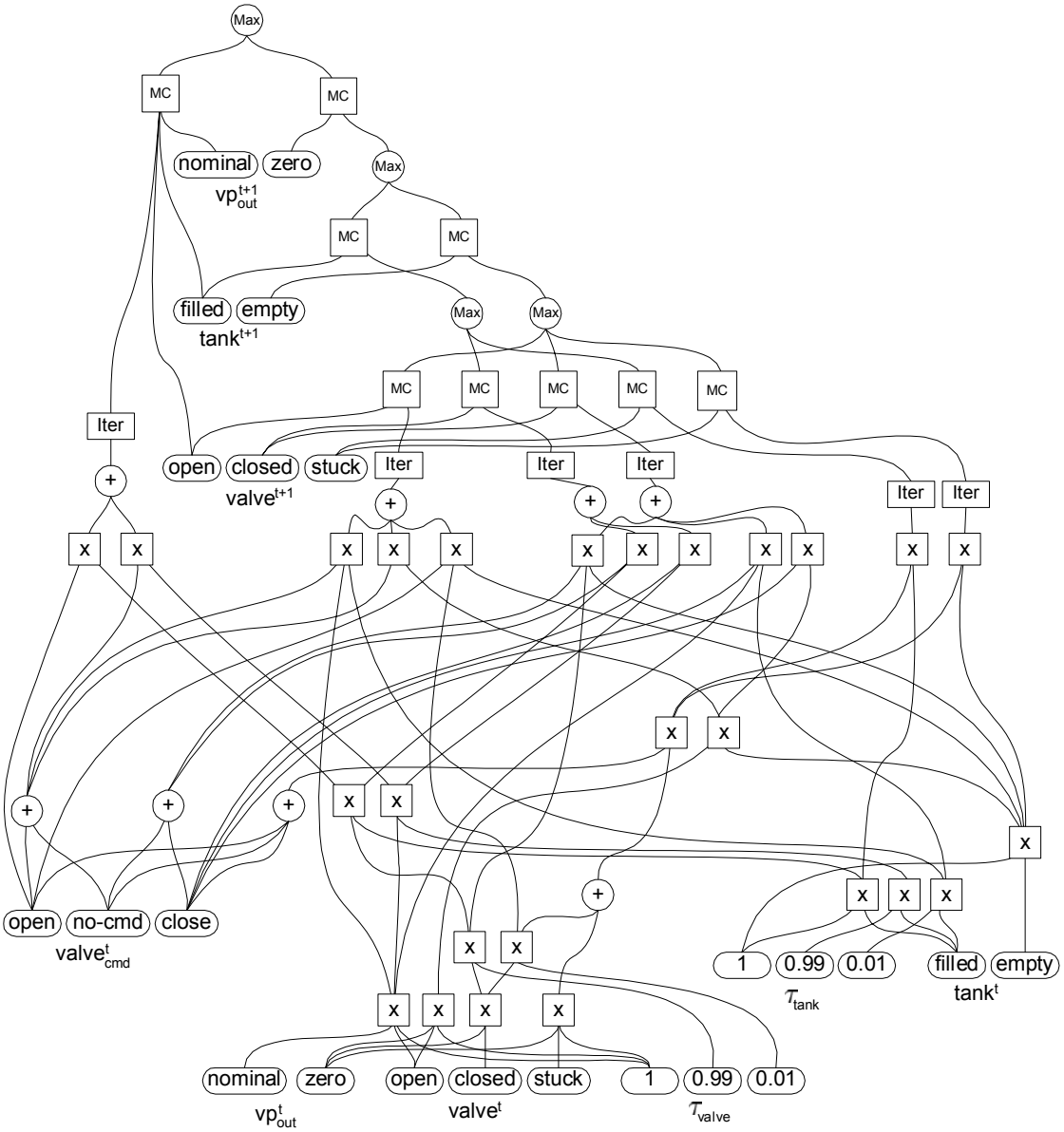

Figure 5-6: This graph computes Eq. 7.36 for the mono-propellant example of this chapter. This graph contains all of the solutions listed in Table 5.1 and is computing Eq. 7.36.

either have only leaves as children, which are all assignments to a variable in $\mathbf{X}_M$, or have only nodes of type $\max^k$ *Combo* as children, and specifically do not have $+$ or $\times$ nodes as children. First recall that smoothness ensures that assignments to the same variables appear under each child of an **Or** node. If the $\max^k$ node has an assignment as a child, then smoothness ensures all of its other children must also be simple assignments. If this is not the case, then there is a child that had assignments to more than one variable in its descendants, violating smoothness. Smoothness also ensures that these assignments are all to the same variable. The $\max^k$ node has the property that it chooses between assignments to the variables $\mathbf{X}_M$, and hence this variable must be from $\mathbf{X}_M$.

A consequence of the first part of this proof is that the $\max^k$ node must have all internal nodes if any child is an internal node. The property of the $\max^k$ node again states that it is choosing between solutions with different assignments to at least one variable in $\mathbf{X}_M$. Let that variable be X_1. By construction, our sd-DNNF chooses between different assignments from $\mathbf{X}_M$ before any other variables. Hence, each child of $\max^k$ corresponds to an assignment to X_1, for example x_1. Thus, independent of what internal node the $\max^k$ has, it can always be re-arranged to have an **And** node as a child and that this **And** node has the assignment x_1 as a child and the remaining sub-tree of the $\max^k$ node as its other child. This sequence, a $\max^k$ node followed by an **And** node followed by an assignment to a variable from $\mathbf{X}_M$, guarantees that the **And** node is labeled as an $\max^k$ *Combo* node, as it involves concatenating an assignment to a variable from $\mathbf{X}_M$ with something else.

A corollary of this property of $\max^k$ nodes is that only $\max^k$ *Combo* nodes have $+$ and $\times$ children. Since the $\max^k$ *Combo* nodes are computing the k best combinations, and otherwise represents multiplication, the $\max^k$ *Combo* nodes naturally integrate the values of $+$ and $\times$ nodes by explicitly multiplying the value of the k best combinations over the $\mathbf{X}_M$ variables by the value computed for each of its $+$ and $\times$ children. Said another way, the $\max^k$ *Combo* node weights the value of all of its solutions over $\mathbf{X}_M$ by the value computed at its $+$ and $\times$ children.

There is still one step left in order to evaluate Eq. 5.1: we need to handle the summation over the variables $\mathbf{X}_\Sigma$, weighted by $\alpha_{\mathbf{x}_\Sigma}$. As we just stated, the $\max^k$ *Combo* nodes naturally represent the point at which values are introduced in the $\arg^{\mathrm{k}} \max_{\mathbf{X}_M}$ problem. Since this point of entry

also corresponds to the bridge between our equations g_i in the tree below, we can introduce an extra operation to be performed at these $\max^k$ *Combo* nodes so as to accommodate this last step. Specifically, if we evaluate the sub-tree(s) of $+$ and $\times$ nodes rooted at an $\max^k$ *Combo* for every g_i and for every $\mathbf{x}_\Sigma$, we can combine them explicitly at the $\max^k$ *Combo* node in accordance with Eq. 5.1. Since $\alpha_{\mathbf{x}_\Sigma}$ just multiplies our value by a constant, we can easily include this in the calculation.

We can express this in another way. At a particular $\max^k$ *Combo* node with $+$ and/or $\times$ children, the $\max^k$ *Combo* implies some (possibly partial) assignment $\mathbf{x}_M$ to the variables $\mathbf{X}_M$. The $+$ and/or $\times$ nodes that are the children of the $\max^k$ *Combo* node are the possible extensions to the assignment $\mathbf{x}_M$ over the remaining variables, given $\mathbf{x}_M$. Note that these extensions are defined in exactly the manner we have been using to compile g_i, and thus the $+$ or $\times$ child of the $\max^k$ *Combo* node represents g_i restricted to the variables of that sub-graph and customized for a particular $\mathbf{x}_M$. If we compile all of our g_i into a single constraint function $\mathcal{C}'$, then the resulting g^* is capable of calculating every g_i, we need only substitute 1 for the value of the variables $\mathbf{X}_f$ introduce for other g_i when evaluating a particular g. Note that we can accomplish this using $h_{\mathbf{x}_{M\Sigma a}}$.

Given that g^* is explicitly rooted at $\max^k$ *Combo*, computing Eq. 5.1 involves computing g_i under $\max^k$ *Combo* for each i and for each of the $\mathbf{x}_\Sigma$ and then computing the product of the g_i values (or their reciprocal) and multiplying by $\alpha_{\mathbf{x}_\Sigma}$. We then accumulate all of these together over all $\mathbf{x}_\Sigma$. Since we are interested in a recursive evaluation of Eq. 5.1, there are in general k $\mathbf{x}_\Sigma$, the k $\mathbf{x}_M$ extracted from the previous evaluation.

The final piece to notice is that every $\max^k$ *Combo* is computing its value for the same g_i and the same $\mathbf{x}_\Sigma$, so we can use dynamic programming for each calculation and share the results among the different $\max^k$ *Combo*; we evaluate the $\times$ and $+$ nodes once per g_i and $\mathbf{x}_\Sigma$.

OCSP Example

To illustrate maximization, consider the left most $\max^k$ *Combo* just under the root max in Fig. 5-6. This $\max^k$ *Combo* corresponds to the assignment $\mathbf{x}_M = \left\{\text{filled}^{t+1}, \text{open}^{t+1}\right\}$, as these are the two children of the $\max^k$ *Combo* node labeled with assignments from $\mathbf{X}_M$. The running example g_i that we have compiled in this chapter, as shown in Fig. 5-5 and Eq. 5.14 corresponds exactly to

this given $\mathbf{x}_M$ and is also the g^* for this example. In Fig. 5-6 we have factored nominal^{t+1} into a separate sub-tree from the rest of Eq. 5.14, reducing Eq. 5.14 to:

$$\begin{aligned} g^*\left(\mathbf{x}_{M\Sigma a}\right) &= f_{\mathbf{x}_{M\Sigma a}}\left(\text{filled}^t\right) \cdot \\ &\Bigg[\Bigg[f_{\mathbf{x}_{M\Sigma a}}\left(\text{open}^t\right) f_{\mathbf{x}_{M\Sigma a}}\left(\text{nominal}^t\right) f_{\mathbf{x}_{M\Sigma a}}\left(\text{tank}_{0.99}\right) f_{\mathbf{x}_{M\Sigma a}}\left(\text{valve}_1\right) \cdot \\ &\qquad \left[f_{\mathbf{x}_{M\Sigma a}}\left(\text{open}^t_{\text{cmd}}\right) + f_{\mathbf{x}_{M\Sigma a}}\left(\text{no-cmd}^t_{\text{cmd}}\right)\right]\Bigg] \\ &+\Bigg[f_{\mathbf{x}_{M\Sigma a}}\left(\text{closed}^t\right) f_{\mathbf{x}_{M\Sigma a}}\left(\text{zero}^t\right) f_{\mathbf{x}_{M\Sigma a}}\left(\text{open}^t_{\text{cmd}}\right) \cdot \\ &\qquad f_{\mathbf{x}_{M\Sigma a}}\left(\text{tank}_1\right) f_{\mathbf{x}_{M\Sigma a}}\left(\text{valve}_{0.99}\right)\Bigg]\Bigg] \end{aligned} \tag{5.15}$$

For reasons of memoization and readability, we have opted to formulate the sub-tree as:

$$\begin{aligned} g^*\left(\mathbf{x}_{M\Sigma a}\right) = &\Bigg[f_{\mathbf{x}_{M\Sigma a}}\left(\text{open}^t_{\text{cmd}}\right) f_{\mathbf{x}_{M\Sigma a}}\left(\text{zero}^t\right) \cdot \\ &f_{\mathbf{x}_{M\Sigma a}}\left(\text{closed}^t\right) f_{\mathbf{x}_{M\Sigma a}}\left(\text{valve}_{0.99}\right) f_{\mathbf{x}_{M\Sigma a}}\left(\text{filled}^t\right) f_{\mathbf{x}_{M\Sigma a}}\left(\text{tank}_1\right)\Bigg] \\ &+\Bigg[\left[f_{\mathbf{x}_{M\Sigma a}}\left(\text{open}^t_{\text{cmd}}\right) + f_{\mathbf{x}_{M\Sigma a}}\left(\text{no-cmd}^t_{\text{cmd}}\right)\right] f_{\mathbf{x}_{M\Sigma a}}\left(\text{nominal}^t\right) \cdot \\ &f_{\mathbf{x}_{M\Sigma a}}\left(\text{open}^t\right) f_{\mathbf{x}_{M\Sigma a}}\left(\text{valve}_1\right) f_{\mathbf{x}_{M\Sigma a}}\left(\text{filled}^t\right) f_{\mathbf{x}_{M\Sigma a}}\left(\text{tank}_{0.99}\right)\Bigg] \end{aligned} \tag{5.16}$$

For our particular example, we are solving the OCSP:

$$\mathbf{X}_a = \left\{\text{valve}^t_{\text{cmd}}\right\} \tag{5.17}$$

$$\mathbf{X}_M = \left\{\text{tank}^{t+1}, \text{valve}^{t+1}\right\} \tag{5.18}$$

$$\mathbf{X}_\Sigma = \left\{\text{tank}^t, \text{valve}^t\right\} \tag{5.19}$$

$$\underset{\mathbf{x}_M \in \mathbb{D}_{\mathbf{X}_M}}{\arg^{\text{k}}\max} \sum_{\mathbf{x}_\Sigma \in \mathbb{D}_{\mathbf{X}_\Sigma}} \frac{g_1\left(\mathbf{x}_M \cup \mathbf{x}_\Sigma \cup \mathbf{x}_a\right)}{g_2\left(\mathbf{x}_M \cup \mathbf{x}_\Sigma \cup \mathbf{x}_a\right)} \cdot \alpha_{\mathbf{x}_\Sigma} \tag{5.20}$$

Where for g_1, $f_x\left(x\right)$ is the transition relation encoded using $\mathbf{X}_f = \left\{\tau_{\text{tank}}, \tau_{\text{valve}}\right\}$ and $\mathbf{X}'_R = \left\{\text{vp}^t_{\text{out}}, \text{vp}^{t+1}_{\text{out}}\right\}$. For g_2, $\mathbf{X}_f = \{\}$, making $f_x\left(x\right)$ irrelevant, and $\mathbf{X}'_R$ is the same.

If we let $\mathbf{x}_M = \left\{\text{filled}^{t+1}, \text{open}^{t+1}\right\}$, $\mathbf{x}_\Sigma = \left\{\text{filled}^t, \text{open}^t\right\}$, and $\alpha_{\mathbf{x}_\Sigma} = 0.5$, then our example

previously has shown that $g_1(\mathbf{x}_M \cup \mathbf{x}_\Sigma \cup \mathbf{x}_a) = 0.99$, and, since all $\mathbf{x}_f$ take on the value 1 for g_2, it is easy to show that $g_2(\mathbf{x}_M \cup \mathbf{x}_\Sigma \cup \mathbf{x}_a) = 1$. We can then compute the value for $\max^k$ *Combo* at the top left, corresponding to our assumed $\mathbf{x}_M$, for this particular $\mathbf{x}_\Sigma$: $\frac{0.99}{1} \cdot 0.5 = 0.495$. If we continued to accumulate values for different $\mathbf{x}_\Sigma$ at this node, we would get some constant p that we can then use to compute $\arg^{\mathrm{k}} \max_{\mathbf{X}_M}$.

5.5 Online Evaluation

Our formulation naturally implies that to solve our OCSP using the compiled representation requires two phases. In the first phase, we compute our sum of products for each g_i for a given $\mathbf{x}_\Sigma$ and combine them together as dictated by our equation. Each g_i can be computed using the g^* rooted at the $\max^k$ *Combo* nodes. We sum together each of these combined values for all of our $\mathbf{x}_\Sigma$, resulting in a single value for each sum-of-products node that is a child of an $\max^k$ *Combo* node. In the second phase, we extract k solutions over $\mathbf{X}_M$, using these computed values to weight each solution. We use the identity value 1 for each assignment to a variable from $\mathbf{X}_M$. The first phase is presented next and the second phase is presented in Chapter 6. This second phase was initially published an Masters of Engineering Thesis [25].

5.5.1 Accumulation Algorithm

For the first phase, we compute:

$$\sum_{\mathbf{x}_\Sigma \in \mathbb{D}_{\mathbf{X}_\Sigma \setminus \mathbf{X}_a}} \left(\prod_{i=1}^{m} g_i(\mathbf{x}_M \cup \mathbf{x}_\Sigma \cup \mathbf{x}_a) \cdot \prod_{i=m+1}^{n} \frac{1}{g_i(\mathbf{x}_M \cup \mathbf{x}_\Sigma \cup \mathbf{x}_a)} \cdot \alpha_{\mathbf{x}_\Sigma} \right) \tag{5.21}$$

for each of the partial assignments to $\mathbf{X}_M$, as compiled into our and-or graph. Since this graph is compiled given that certain variables are always part of $\mathbf{X}_a$, Eq. 5.21 can only be computed given an $\mathbf{x}_a$ with assignments to these variables. We have also assumed that the number of $\alpha_{\mathbf{x}_\Sigma}$ with non-zero values is small, so we compute Eq. 5.21 by explicitly summing over each $\mathbf{x}_\Sigma$ and each g_i on the compiled representation.

Algorithm 5.1: Accumulate(V_O, E, $\mathcal{L}_{\mathbf{P}}$, $\mathcal{L}_L$, $\times$, $+$, $\mathbf{x}_a$, S, G, m)

```
1  foreach v ∈ V_MC do p*(v) ← 0 ;
2  for j = 1 to |S| do
3      x_Σ ← S(j).x_Σ ;
4      α_{x_Σ} ← S(j).α_{x_Σ} ;
5      foreach v ∈ V_MC do p(v) ← α_{x_Σ} ;
6      for i = 1 to |G| do
7          f_{x_{MΣa}} ← (G(i)) (L_P ∘ L_L^{-1}, x_a, x_Σ) ;
8          P_V ← g*(V_O, E, L_L, ×, +, f_{x_{MΣa}}) ;
9          if i ≤ m then
10             foreach v ∈ V_MC do p(v) ← p(v) × P_V(v) ;
11         else
12             foreach v ∈ V_MC do p(v) ← p(v)/P_V(v) ;
13         end
14     end
15     foreach v ∈ V_MC do p*(v) ← p*(v) + p(v) ;
16 end
17 return p* ;
```

The top-level algorithm for computing Eq. 5.21 is shown in Alg. 5.1. The parameters to Accumulate are:

- V_O: An ordering of the and-or nodes such that each node appears before its children, and thus the leaves are towards the end. In this case we assume V_O is an ordering that only includes those leaves labeled with assignments to $\mathbf{X}_\Sigma \cup \mathbf{X}_R \cup \mathbf{X}_a$, the $\times$ nodes, the $+$ nodes, and those $\max^k$ *Combo* nodes that have one or more of these other nodes as a child. This list of nodes are the only ones that pertain to Eq. 5.21.

- E, $\mathcal{L}_{\mathbf{P}}$, $\mathcal{L}_L$, $\times$, $+$: The other terms that define the compiled and-or graph.

- $\mathbf{x}_a$: The assignment that pertains to our whole problem and that we are thus to use when computing Eq. 5.21.

- S: A list of assignments $\mathbf{x}_\Sigma$ and corresponding values $\alpha_{\mathbf{x}_\Sigma}$ that fully specify our summation $\sum_{\mathbf{X}_\Sigma \setminus \mathbf{X}_a}$. We assume in our algorithm that any $\mathbf{X}_\Sigma \cap \mathbf{X}_a$ are consistent with respect to $\mathbf{x}_\Sigma$ and $\mathbf{x}_M$, though this need not be assumed. Inconsistent values can be caught and set to 0.

- G, m: Terms that specify each g_i. G specifies a list of functions, m of which are products and $(|G| - m)$ of which are reciprocals. Each function of G takes the set of innate values f_x specified by $\mathcal{L}_{\mathbf{P}}$ and the current $\mathbf{x}_a$ and $\mathbf{x}_\Sigma$. It returns the function $f_{\mathbf{x}_{M\Sigma a}}$, suitable for evaluating a particular g_i using g^*. Since $f_{\mathbf{x}_{M\Sigma a}}$ evaluates to a value for each of its input assignments, a function of G effectively returns a vector of values, one for each assignment.

Alg. 5.1 computes Eq. 5.21 for each $\max^k$ *Combo* node and stores this value in the variable p^*, which is indexed by the $\max^k$ *Combo* node. On Line 1, this value is initially set to 0 for all $\max^k$ *Combo* nodes. On lines 2-16, the algorithm is looping over the different values $\mathbf{x}_\Sigma$ as specified in S. Within this loop, the algorithm begins by retrieving the assignment $\mathbf{x}_\Sigma$ and its value $\alpha_{\mathbf{x}_\Sigma}$ from S, on lines 3 and 4, respectively. The algorithm then sets a variable p to $\alpha_{\mathbf{x}_\Sigma}$ on Line 5 for each $\max^k$ *Combo* node, where p is counting the value added to the $\max^k$ *Combo* node by the value $\mathbf{x}_\Sigma$. We are here through Line 14 computing

$$p = \left(\prod_{i=1}^{m} g_i \left(\mathbf{x}_M \cup \mathbf{x}_\Sigma \cup \mathbf{x}_a \right) \cdot \prod_{i=m+1}^{n} \frac{1}{g_i \left(\mathbf{x}_M \cup \mathbf{x}_\Sigma \cup \mathbf{x}_a \right)} \cdot \alpha_{\mathbf{x}_\Sigma} \right) \tag{5.22}$$

where we have initialized p to $\alpha_{\mathbf{x}_\Sigma}$.

Lines 6-14 are computing the g_i. Line 7 retrieves the set of values for $f_{\mathbf{x}_{M\Sigma a}}$ from G given $\mathbf{x}_a$ and $\mathbf{x}_\Sigma$. Line 8 evaluates Alg. 5.2 below for the particular g_i. This evaluation returns the value of g_i for each $\max^k$ *Combo* node in P_V. Lines 9-13 incorporate each value in P_V into p by either multiplying or dividing, depending on the g_i. The algorithm then adds these resulting p values into p^* on Line 15. Finally, these per-$\max^k$ *Combo* values p^* are returned on Line 17.

The next section shows how to compute g^*.

5.5.2 g^* Algorithm

The basic idea for g^* is to evaluate each node as appropriate to its type. For instance, leaves are evaluated using $f_{\mathbf{x}_\Sigma}$, while **And** nodes are evaluated using $\times$. The structure of the and-or graph ensures that evaluating the graph properly computes g^*. Alg. 5.2 uses dynamic programming to compute g^*. In this algorithm, each child is evaluated before its parent, ensuring that a parent can

Algorithm 5.2: $g^*(V_O, E, \mathcal{L}_L, \times, +, f_{\mathbf{x}_{M\Sigma a}})$

```
for i = |V_O| to 1 do
    v ← V_O [i] ;
    switch v in
        case v ∈ L
            P_V (v) ← f_{x_{MΣa}} (L_L (v)) ;
        end
        case v ∈ A
            P_V (v) ← 1 ;
            foreach ⟨v, n⟩ ∈ E do
                P_V (v) ← P_V (v) × P_V (n) ;
            end
        end
        case v ∈ O
            P_V (v) ← 0 ;
            foreach ⟨v, n⟩ ∈ E do
                P_V (v) ← P_V (v) + P_V (n) ;
            end
        end
    end
end
return P_V ;
```

compute its value based on all cached values at each of its children.

The main loop of the algorithm, lines 1-20 are looping over all of the nodes in the and-or tree that are involved in computing g^*, namely the leaves for the variables $\mathbf{X}_\Sigma \cup \mathbf{X}_R \cup \mathbf{X}_a$, the $\times$ nodes, the $+$ nodes, and the $\max^k$ *Combo* nodes with one or more of the other nodes as its child. Each node either corresponds to a leaf, an **And** node, or an **Or** node, and this algorithm evaluates the node based on its type. The node being evaluated is specified by Line 2. For leaf nodes, the value of a leaf is $f_{\mathbf{x}_\Sigma}$ applied to the assignment of the leaf, as shown on Line 5. For **And** nodes, the value of the node is the product of all of its children. The algorithm initially sets the node's value to the multiplicative identity 1 and then loops over each child, multiplying its value by the child's value, on Line 10. For **Or** nodes, the vale of the node is the sum of all of its children. The algorithm initially sets the node's value to the additive identity 0 and then loops over each child, adding its child's value to its own, on Line 16. On Line 21, the computed values for all nodes P_V is returned.

5.5.3 Runtime Analysis

We start with the g^* algorithm. We assume that accessing the nodes using the ordering is an $O(1)$ operation. The outer loop is $|V_O|$ iterations. Within this loop, the algorithm accesses every edge once from the parent's side, which is thus $|E|$ accesses. The algorithm computes $\mathcal{L}_L$ once for each leaf node and $f_{\mathbf{x}_\Sigma}$ once as well. Since we assume both of these are just structure or indexed array lookups, these are $O(1)$ operations. We also assume the value of the node is stored with the node, so $P_V\,(v)$ is an $O(1)$ operation. We assume native multiplication and addition, and so, while we compute $O(|E|)$ multiplications and additions, these do not incur more than an $O(|E|)$ cost. We return P_V by returning the values on the nodes. Since $|V_O| \leq (|E|+1)$, this algorithm as an $O(|E|)$ time complexity. Based on this discussion, we store a value per node for P_V, in addition to the graph, so this algorithm requires an additional $O(|V|)$ storage space for values in addition to the space used by the graph.

We now move on to Accumulate, Alg. 5.1. If we assume we have a list of pointers to each $\max^k$ *Combo* node and that we store all values with the $\max^k$ *Combo* node, then the algorithm initially uses $O(|MC|)$ time and space to initialize p^*. The algorithm then loops over each assignment in

S, where in general we assume $|S| \leq k$, the argument of our arg max. For each of these loops the algorithm performs two $O(1)$ calculations and then an $O(|MC|)$ time and space initialization of p.[2] The algorithm then evaluates each g_i, of which there are $|G|$. We evaluate each element of G once to compute $f_{\mathbf{x}_\Sigma}$, with a complexity that depends on our g_i. It is in general at least $\Omega(|L|)$, and is $O(|L|)$ for the problems we have considered, using indexed array lookups. The algorithm then evaluates g^* and either multiplies or divides by the result for each $\max^k$ *Combo* node. At the end of the loop, the algorithm performs one addition per MC node. We assume the algorithm returns the values p^* through the nodes. Since computing g^* in the inner loop dominates the time complexity of the inner loop, for our problems, with a complexity of $O(|E|)$, the inner loop has a time complexity of $O(|G||E|)$. This in turn dominates the outer loop's time complexity, yielding a total time complexity of $O(|S||G||E|)$. This algorithm requires an additional $O(|MC|)$ space.

The next chapter shows how to use the values computed in p^* to extract the k maximal solutions. In the next chapter, we treat each of these values computed as a child of the MC node with the label $\emptyset$, though in practice these $\emptyset$ nodes need not be explicitly added to the graph. This value, using either approach, just multiplies all the solutions of the $\max^k$ *Combo* node by a constant. The next section shows how to sample single solutions from the distribution we just computed with Alg. 5.1.

5.6 OCSP Sampling

In this section we introduce the notion that our representation, after running the accumulate algorithm, Alg. 5.1, encodes the full valuation of each $\mathbf{x}_M$. For the observation sampling portion of the previous chapter, in Section 4.2.4, we sample from this full distribution rather than extract just k answers. The algorithm for extracting the k best answers naturally builds upon the notion that the graph contains the full valuation of each $\mathbf{x}_M$ as presented in the next chapter. We thus in this section introduce how to sample from this distribution and then in the next chapter we show how to extract the k best solutions.

Specifically we have an sd-DNNF with the leaves labeled with some set of assignments to $\mathbf{X}_M$

[2] If we combine Algs. 5.1 and 5.2, we can immediately incorporate the value computed by g^* into p instead of first storing it in P_V, eliminating the space cost of this algorithm, though not the time.

and a set of **And** nodes, some of which have associated values calculated using Alg. 5.1. We can sample from this valued sd-DNNF with two insights: first, we have the insight that we can compute the total value in the sd-DNNF by evaluating the sd-DNNF using $+$ for **Or** nodes and $\times$ for **And** nodes. The value accumulated at the root is the total value of the sd-DNNF. Second, we have the insight that each **Or** node can use the accumulated value of each of its children to proportionally choose among them given a random number r, specifying a particular selection. Using these two insights, we generate a sample by generating a random selection and reading off the resulting solution.

If we treat the **And** node values specified by p^* as leaf children of the **And** node labeled with $\emptyset$, then the first step is equivalent to invoking $g^*\left(V_O, E, \mathcal{L}_L, \times, +, f_{\mathbf{x}_{M\Sigma a}}\right)$ with:

$$f_{\mathbf{x}_{M\Sigma a}}(\mathbf{x}) = \begin{cases} p^* \text{from Alg. 5.1} & \text{If } \mathbf{x} \text{ is an } \textbf{And} \text{ node probability.} \\ 1 & \text{Otherwise} \end{cases} \tag{5.23}$$

except we keep the accumulated values on all of the nodes, not just the $\max^k$ *Combo* nodes.

The second step of the algorithm uses a depth-first search to sample a selection. At leaf nodes, the algorithm adds the label of the leaf to the sample. At **And** nodes, the algorithm recurses on all of its children. Finally, at **Or** nodes, the algorithm chooses a random number and selects the corresponding child using Alg. 4.5, having normalized its children's values to sum to 1.

Algorithm 5.3: DNNFSample(V_O, E, $\mathcal{L}_L$, p^*)

1 $f_{\mathbf{x}_{M\Sigma a}} \leftarrow$ Eq. 5.23 using p^* ;
2 $P_V \leftarrow g^*\left(V_O, E, \mathcal{L}_L, \times, +, f_{\mathbf{x}_{M\Sigma a}}\right)$;
3 $v \leftarrow V_O\left[1\right]$;
4 **return** *SampleWithVals*$(v, V_O, E, \mathcal{L}_L, P_V, \{\})$;

The top-level algorithm, Alg. 5.3, invokes these two sub-routines so as to generate a sample. On Line 1, the algorithm creates $f_{\mathbf{x}_{M\Sigma a}}$ according to Eq. 5.23 above and then invokes g^* using this $f_{\mathbf{x}_{M\Sigma a}}$ on Line 2. g^* returns the accumulated values on all the nodes, representing the relative value of each node as compared to its siblings. Then, the algorithm gets the root node of the sd-DNNF on Line 3 and calls SampleWithVals on Line 4. We pass in the root node as the current node in the

recursion and the empty solution as the solution $\mathbf{m}$.

Sample With Values

Algorithm 5.4: SampleWithVals$(v, V_O, E, \mathcal{L}_L, P_V, \mathbf{m})$

```
1  switch v in
2      case v ∈ L
3          m ← m ∪ {L_L(v)} ;
4      end
5      case v ∈ A
6          forall ⟨v, n⟩ ∈ Edges do
7              m ← SampleWithVals(n, V_O, E, L_L, P_V, m) ;
8          end
9      end
10     case v ∈ O
11         A ← { ⟨n, p⟩ | p = P_V(n) / P_V(v) } ;
12         n ← Sample from A using SampleBeliefState ;
13         m ← SampleWithVals(n, V_O, E, L_L, P_V, m) ;
14     end
15 end
16 return m ;
```

For this sub-routine, we are choosing a random selection based on the accumulated values at each node. Recall that there is a one-to-one correspondence between selections and solutions. For leaf nodes, on Line 3, the algorithm adds the leaf's label to the solution. For **And** nodes, the algorithm gathers a random selection rooted at each of its children, on lines 6-8. For **Or** nodes, we first note that $P_V(v)$ is the sum of the value of all of v's children. We can thus use $P_V(v)$ when constructing the array A to normalize the values of all of v's children so they sum to 1 on Line 11. We normalize A so we can use SampleBeliefState on Line 12 to sample a node based on the distribution in A. The algorithm then recursively uses this sampled node as part of the selection and recursively gathers the rest of the selection on Line 13.

5.6.1 Runtime Analysis

As we saw earlier in this chapter, the time complexity of g^* is $O(|E|)$. Alg. 5.4 visits each node in the graph at most once, since the algorithm only branches on **And** nodes and the children of **And** nodes do not share the same variables, so they also do not share the same nodes. For **Or** nodes, the algorithm considers each edge once as part of the sampling. Thus, the complexity of algorithm is equal to the number of nodes in a selection plus the number of edges connected to the or nodes in the selection. Since this is always going to be less than $|E|$, the complexity of Alg. 5.3 is dominated by g^* and is also $O(|E|)$.

5.7 Conclusion

This chapter has shown how to compile our Optimal Constraint Satisfaction Problem into an compact representation suitable for solving the problem, up to the point of extracting the k best solutions. This compilation process involved first reducing our problem to an OCSP by introducing additional variables that capture the cost functions f_j and then compiling the resulting equations into an and-or graph. This graph is extended from a traditional valued sd-DNNF, using summation and multiplication, into a valued and-or graph that includes nodes for maximization and maximal combinations. By introducing these additional nodes, this chapter presented the additional rules required to properly compile this extended problem and still be able to correctly evaluate the and-or tree. This chapter then presented how to evaluate this and-or tree to find solutions to the OCSP. This chapter presented the g^* algorithm that computes the summation and multiplication of the OCSP. This algorithm requires $O(|S||G||E|)$ time and $O(|V|)$ space. The next chapter shows how to use the results of g^* to extract the solutions to the OCSP.

This chapter also presented an algorithm that uses g^* and a sampling algorithm in order to generate random samples from the distribution generated by g^*.

Chapter 6

K-Best-Solutions Algorithm

In Chapter 5, we showed how to reduce

$$\underset{\mathbf{x}_{M\setminus a}\in\mathbb{D}_{\mathbf{x}_M\setminus\mathbf{x}_a}}{\arg^{\mathrm{k}}\max} \sum_{\mathbf{x}_{\Sigma\setminus a}\in\mathbb{D}_{\mathbf{x}_\Sigma\setminus\mathbf{x}_a}} \frac{\prod_{i=1}^{m} g_i\left(\mathbf{x}_{M\setminus a}\cup\mathbf{x}_{\Sigma\setminus a}\cup\mathbf{x}_a\right)}{\prod_{i=m+1}^{n} g_i\left(\mathbf{x}_{M\setminus a}\cup\mathbf{x}_{\Sigma\setminus a}\cup\mathbf{x}_a\right)}\cdot\alpha_{\mathbf{x}_\Sigma} \tag{6.1}$$

down to a maximization of products by applying the Accumulate algorithm. This maximization of products is encoded in a valued sd-DNNF $\langle V, E, \mathcal{L}_L, \mathcal{L}_{\mathbf{P}}, \times, \max\rangle$, and the purpose of this chapter is to extract the k best solutions to the Valued CSP encoded by the sd-DNNF.

Other approach to solving this problem exist, such as Sachenbacher [49]. The general assumption by such algorithms is that the search-graph is created on the fly, so they use some form of decomposition scheme to explore the space as needed. In these approaches, the assignment associated with each choice is available on the choice arc. For our problem, the graph already exists, and the assignments implied by a choice exist on the leaves of the subgraph associated with the choice. This chapter contributes an algorithm for extracting solutions in this circumstance of having an explicit graph with labels at the leaves as well as an improved algorithm for generating solutions at **And** nodes. The latter improvement is explained in Section 6.3.2.

This chapter builds a best-solution algorithm that extracts only one solution ($k = 1$), presented

again in Appendix A. The basic principle of the k-best-solutions algorithm is to record at each node in the graph the k best selections rooted at that node. When propagated up to the root node, the root node will then have the k best selections overall. The selections can then be used to extract the k best solutions.

This information requires significantly more space to record than just the best selection for or nodes η used in Appendix A, specifically we need to record selection information for both **And** and **Or** nodes, as each node chooses among the k best selections of each child. We extend our three rules for specifying the best selection from page 185 to support k selections:

Rule 1: For each leaf node l, the value of the leaf node is $\mathcal{L}_{\mathbf{P}}(l)$.

Rule 2: For each **And** node a, we want the k best combinations of its children, where a combination includes a selection from each child of a and the value of the combination is computed by applying $\times$ to the value of the selections of the children. Assume that a has p children, v_1, ..., v_p. Each child has between 1 and k selections recorded. If we denote the selections of a child as $\text{Sel}(v_i)$, then there are $\prod_{i=1}^{p} |\text{Sel}(v_i)|$ combinations. The value of a combination is computed by using $\times$ to combine the values of each child's selection in the combination. The k best selections for the parent a are the k best combinations, ordered by $\max$.

Rule 3: For each **Or** node o, a selection of o chooses only one selection from one child. To generate the k best child selections, we consider all the selections of the p children of o and choose the k best selections among all of them, ordered by $\max$.

We start with an example of Rule 2. For our example, the **And** node a has 2 children, v_1 and v_2, and we have $k = 3$. Node v_1 has 3 selections, with values 0.3, 0.2, and 0.1. Node v_2 has 2 selections, with values 0.5 and 0.2. We denote a combination as $\langle i, j \rangle$, where we have numbered $\text{Sel}(v_i)$ from 1 to k, and so i is the ith selection of v_1 and j is the jth selection of v_2. There are 6 combinations of the selections of a: $\langle 1,1 \rangle$, $\langle 1,2 \rangle$, $\langle 2,1 \rangle$, $\langle 2,2 \rangle$, $\langle 3,1 \rangle$, and $\langle 3,2 \rangle$. Assuming multiplication for $\times$, the values of these combinations are 0.15, 0.06, 0.1, 0.04, 0.05, and 0.02, respectively. The 3 best combinations are 0.15, 0.1 and 0.06, since $k = 3$, and correspond to $\langle 1,1 \rangle$, $\langle 2,1 \rangle$, and $\langle 1,2 \rangle$, respectively. The selections variable $\text{Sel}(a)$ is set to these three combinations.

Now consider a similar example of Rule 3, for an **Or** node o. The node has 2 children, v_1 and v_2, where v_1 has 3 selections and v_2 has 2 selections. We let $k = 3$. We use the values 0.3, 0.2, and 0.1 for v_1's selections, and 0.2 and 0.1 for v_2's selections. The top 3 selections for o are selection 1 and 2 of v_1 and selection 1 of v_2, hence Sel (o) is set to these three selections. Note that we assume a total ordering with $\max$, hence, for example, we can us the index of v_i in V_O to break ties when the values are equal.

Rules 2 and 3 both require a value for each of their child's selections, hence we extend the value recording variable $\mathbf{P}_V(v)$ of Appendix A to include both the node and the selection: $\mathbf{P}_V(v, i)$. We need a way to know how many selections a node has, as it may be less than k, hence we define a new function #Sel that returns the number of selections of a node: #Sel (v).

As with Appendix A, we generate a modified graph that describes the k best selections. In the modified graph, every node is effectively replicated once for every selection it records, thus nodes are indexed by the sd-DNNF's node and the selection number. We denote this $\langle v, j\rangle$ for a node v and the jth selection. A leaf always has exactly 1 selection, hence for a leaf node l, our modified sd-DNNF graph has the one node $\langle l, 1\rangle$.

To efficiently extract the k best solutions, given that we have computed the k best selections, we again augment the information recorded by the algorithm of Appendix A. For each selection of an **Or** node o, we need to know which child's selection was chosen for each of o's selections. We thus extend η to be a function $O \times \{1, \ldots, k\} \to V \times \{1, \ldots, k\}$, where we constrain $\eta(o, i) = \langle v, j\rangle$ such that $v \in$ Children (o), $i \leq$ #Sel (o), and $j \leq$ #Sel (v). This function connects selections of o to selections of v.

For **And** nodes, we now need to know which combination of its children was chosen for each of the **And** node's selections. We record this information in ξ. Recall that for an **And** node a, the ith best selection of a is a combination of the selections of the children of a. We define ξ as the function $A \times \{1, \ldots, k\} \times V \to \{1, \ldots, k\}$, where we constrain $\xi(a, i, v) = j$ such that $v \in$ Children (a), $i \leq$ #Sel (a), and $j \leq$ #Sel (v). ξ records the selection of the child v corresponding to the ith best selection of a, specifically the selection $\langle v, j\rangle$. In Appendix A, where $k = 1$, a and every child of a has only one selection. There is only one combination possible when $k = 1$; this is the first and

only selection of every child of a. When $k = 1$, ξ is unnecessary as the i and j above are always 1, and thus ξ is unnecessary in Appendix A.

To illustrate ξ, lets look back at the **And** node example given above, where a has two children, v_1 and v_2. The best 3 selections are $\langle 1, 1\rangle$, $\langle 2, 1\rangle$, and $\langle 1, 2\rangle$, in that order. Then for this fixed a, $\xi(a, i, v)$ defines a 3×2 table:

		v	
		v_1	v_2
	1	1	1
i	2	2	1
	3	1	2

where, for instance, the entry at $(3, v_2)$ has a value of 2, the value of $\xi(a, 3, v_2)$. This entry means that $\langle a, 3\rangle$ is connected to $\langle v_2, 2\rangle$. $\langle a, 3\rangle$ is also connected to $\langle v_1, 1\rangle$.

The fully computed η and ξ functions define up to k selections, where the number of selections defined is the number of selections of the root node, #Sel (r). Given up to k selections, defined by η and ξ, we extract the corresponding solutions. As in Appendix A, the solutions are defined by paths from the root to the leaves, in the graph modified by η and ξ. The ith solution is the set $\mathcal{L}_L(l)$, where l consists of all the leaves that have a path from the ith selection of the root node. A path for the ith selection of the root starts at the node $\langle r, i\rangle$. For every **And** node $\langle a_j, i_1\rangle$ along the path at position j, the node $\langle v_{j+1}, i_2\rangle$ at position $j + 1$ in the path must be such that $\xi(a_j, i_1, v_{j+1}) = i_2$. That is to say that $\langle a_j, i_1\rangle$ connects to $\langle v_{j+1}, i_2\rangle$ in the modified graph. For every **Or** node $\langle o_j, i_1\rangle$ along the path at position j, the node $\langle v_{j+1}, i_2\rangle$ at position $j + 1$ in the path must be such that $\eta(o_j, i_1) = \langle v_{j+1}, i_2\rangle$. That is to say that $\langle o_j, i_1\rangle$ connects to $\langle v_{j+1}, i_2\rangle$ in the modified graph.

We extend the notion of marking developed in Appendix A by noting that each of a node's k selections may be part of any subset of the k root selections, but that each root selection i includes either one or zero of a node's k selections. This fact implies that we can most efficiently represent the relationship between a node's k selections and the k root selections by recording which of the node's k selections corresponds to a particular root selection, if any, rather than recording which root selections, if any, correspond to a node's selection. There is never more than one of a node's

selections included in the root selection, as we noted before, due to the decomposition property of an sd-DNNF; a selection necessarily forms a tree in the modified graph, including each node at most once. For each node, rather than storing just a marking as before, we store k markings, indexed by the root selection. Each marking $m(v,i)$ either takes on the special "none" value $\perp$ or a value j, which specifies the jth selection of the node v is part of the ith root selection. Initially all the marks are $\perp$, except the root markings, for which $m(r,i) = i$ for each i from 1 to #Sel(r). To extract the k solutions, we walk over the original sd-DNNF from the root to the leaves, propagating the k markings to the children.

For an **And** node a, for each i such that $m(a,i) \neq \perp$, and for each child v of a, we set $m(v,i) = \xi(a, m(a,i), v)$. Recall that if $m(a,i) = j_1$ then $\langle v, j_1 \rangle$ is part of the ith root selection. If $\xi(a, j_1, v) = j_2$, then $\langle v, j_2 \rangle$ is part of the j_1^{th} best selection of a and thus the above records that $\langle v, j_2 \rangle$ should also be part of the ith solution, just like $\langle a, j_1 \rangle$.

For an **Or** node o and for each i such that $m(o,i) = j_1$, $j_1 \neq \perp$, if $\langle v, j_2 \rangle = \eta(o, j_1)$, then the selection $\langle o, j_1 \rangle$ is part of the ith root selection and the selection $\langle v, j_2 \rangle$ is also part of the ith root selection because it is part of the j_1^{th} selection of o. We thus set $m(v,i) = j_2$.

For a leaf node l, $m(l,i)$ is either $\perp$ or 1, as the leaf always has exactly one selection. Thus, for each $m(l,i) = 1$, the ith solution includes the symbol of l, $\mathcal{L}_L(l)$.

For the rest of this chapter, we first present a example used throughout the rest of this chapter and then present the algorithm that solves the problem of finding the k best solutions.

6.1 A-B Example

We will use a contrived example with two types of labels, A and B, to illustrate the algorithms throughout this chapter. Each type of label has three values, for example a1, a2, and a3. The truth table for these two variables is:

	a1	a2	a3
b1	1	1	0
b2	0	1	1
b3	0	0	1

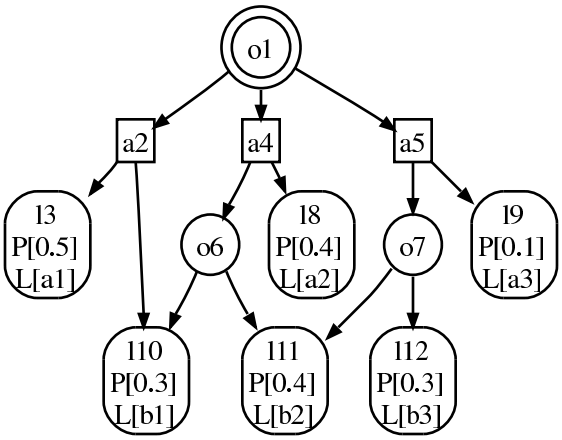

Figure 6-1: This figures shows a valued sd-DNNF corresponding to the A-B constraints. This graph has 5 solutions: {“a1”, “b1”}, {“a2”, “b1”}, {“a2”, “b2”}, {“a3”, “b2”}, and {“a3”, “b3”}.

The sd-DNNF shown in Fig. 6-1 represents this truth table. There are six leaves, three of which are l-3, l-8, and l-9 with labels “a1”, “a2”, and “a3” from A, respectively. The other three are l-10, l-11, and l-12 with labels “b1”, “b2”, and “b3” from B, respectively. We now drop the “-” and use l10 instead of l-10. The complete list of nodes is: o1, a2, l3, a4, a5, o6, o7, l8, l9, l10, l11, and l12. The values $\mathcal{L}_{\mathbf{P}}$ of these six leaves are $\mathcal{L}_{\mathbf{P}}(\text{l3}) = 0.5$, $\mathcal{L}_{\mathbf{P}}(\text{l8}) = 0.4$, $\mathcal{L}_{\mathbf{P}}(\text{l9}) = 0.1$, $\mathcal{L}_{\mathbf{P}}(\text{l10}) = 0.3$, $\mathcal{L}_{\mathbf{P}}(\text{l11}) = 0.4$, and $\mathcal{L}_{\mathbf{P}}(\text{l12}) = 0.3$. There are 13 edges E, all of which are drawn in Fig. 6-1 and are thus omitted from this textual description. We are assuming a maximum-product for $\max$ and $\times$, respectively.

We now present an algorithm to compute the k best solutions. The sub-routines of this algorithm have the hierarchy shown in Fig. 6-2.

6.2 Find-K-Best-Solutions Algorithm

Algorithm 6.1: FindKBestSolutions(V_O, E, $\mathcal{L}_L$, $\mathcal{L}_{\mathbf{P}}$, $\times$, $\max$, k)

1 $\langle \eta, \xi, \#_r \rangle \leftarrow$ FindKBestSelections(V_O, E, $\mathcal{L}_{\mathbf{P}}$, $\times$, $\max$, k) ;
2 $\mathcal{S}_k \leftarrow$ GetKSolutionsFromSelections(V_O, E, $\mathcal{L}_L$, η, ξ, $\#_r$) ;
3 **return** $\mathcal{S}_k$;

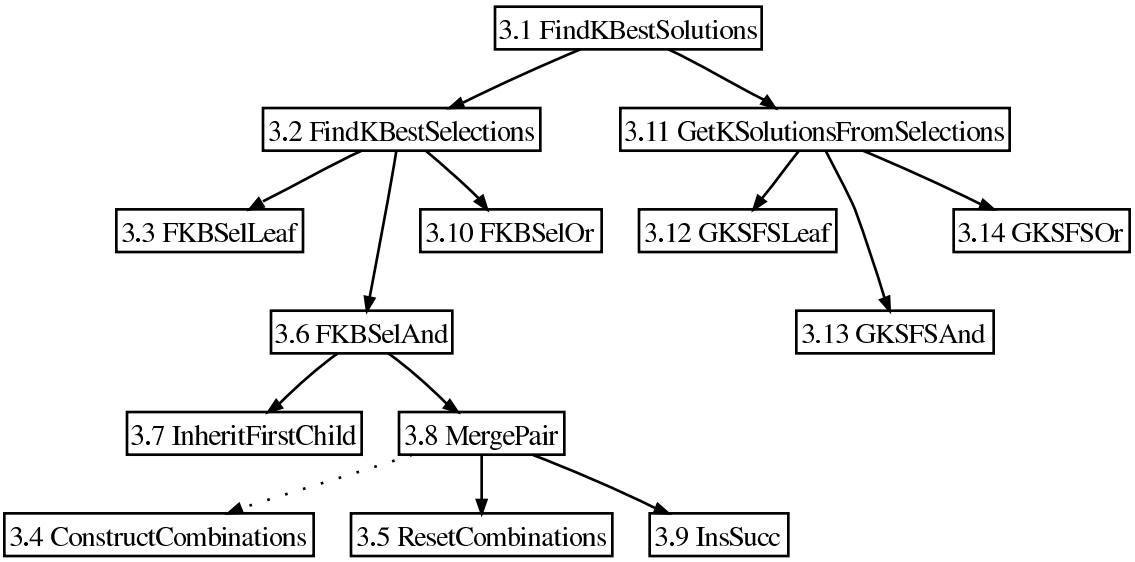

Figure 6-2: This diagram shows how the various algorithms of this chapter are related. The top-level algorithm is FindKBestSolutions. The ConstructCombinations function is the only unusual item in this diagram, as it is expected that it is run prior to running FindKBestSolutions so that its output can be used by MergePair. ConstructCombinations only depends on k.

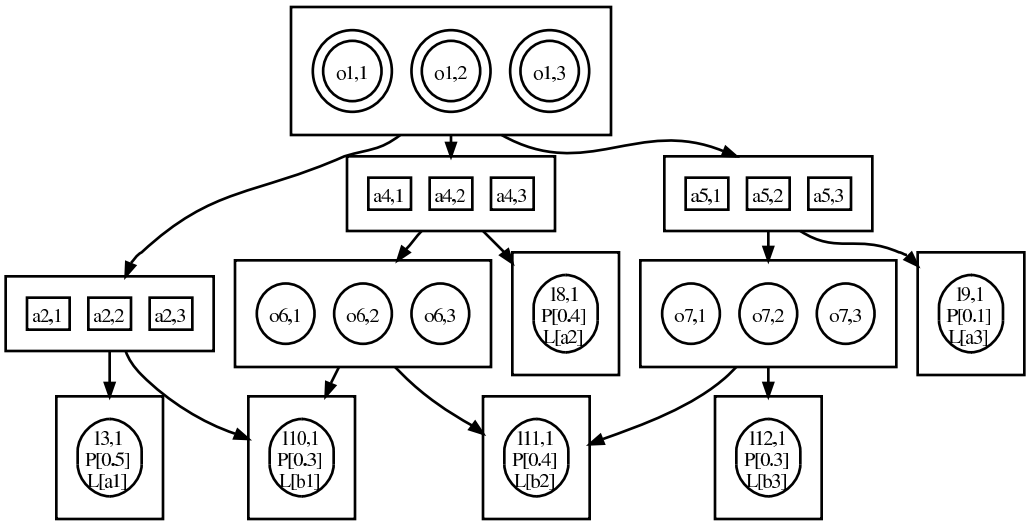

Figure 6-3: This figure shows the modified nodes of the A-B example when $k = 3$.

As with the $k = 1$ algorithm, we break this algorithm down into two parts. The first part makes a pass from the leaves to the root, computing η and ξ. This involves internally computing up to k values per node. Together, η and ξ define between 1 and k selections rooted at each internal node. The value $\#_r$ specifies the number of selections defined by η and ξ, between 1 and k. The second part of the algorithm makes a pass from the root to the leaves, extracting the $\#_r$ best solutions from the selections. These are then returned.

Alg. 6.1 is generating a graph on top of the existing graph, described by η and ξ, that has up to k replicas of each internal node, and a copy of a subset of the edges of the duplicated node. This algorithm then extracts up to k solutions. We denote each of these duplicate graph nodes $\langle v, i \rangle$. Fig. 6-3 shows the duplicate, "modified" nodes of Fig. 6-1 explicitly for $k = 3$. Initially, when Alg. 6.1 begins, all of these modified nodes exist but have no edges. The objective of Alg. 6.2 is to add the appropriate edges in ξ and η to define the best selections at each internal node.

For this example, when $k = 3$, some modified nodes never have edges because there are less than 3 selections rooted at the unmodified node. For example, a2 has only one selection, which includes itself, l3 and l10. We thus omit these nodes from Fig. 6-3 to save space, shown in Fig. 6-4.

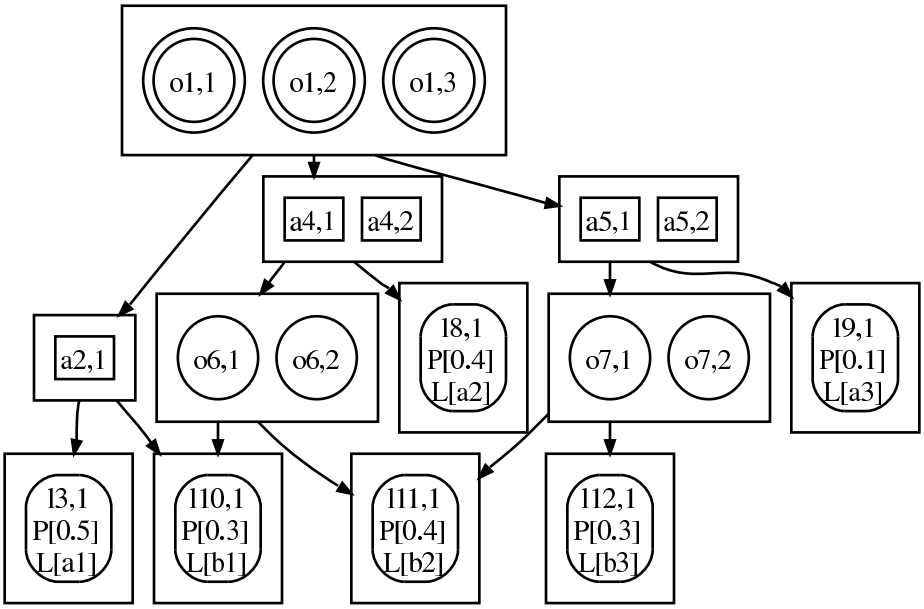

Figure 6-4: This figure shows just the modified nodes of the A-B example, when $k = 3$, that have an edge after running FindKBestSolutions, Alg. 6.1. These nodes are unnecessary and need not be allocated.

Fig. 6-4 represents the starting point of Alg. 6.1.

For $k = 3$, Alg. 6.1, FindKBestSolutions, calls Alg. 6.2, FindKBestSelections, which generates the edges shown in Fig. 6-5. Due to space constraints, we have abbreviated ξ by omitting the last term n in $\xi\,(a, j, n)$, as the last term is the node to which the arc points. Alg. 6.1 then calls Alg. 6.11, GetKSolutionsFromSelections, which extracts the three solutions of the selections rooted at o1. The three solutions are, in order, {"a2", "b2"}, {"a1", "b1"}, and {"a2", "b1"}. The first two selections are highlighted in Fig. 6-12 on page 144. The third selection overlaps with the first two, so it is not drawn.

6.3 Find-K-Best-Selections Algorithm

Algorithm 6.2: FindKBestSelections(V_O, E, $\mathcal{L}_{\mathbf{P}}$, $\times$, $\max$, k)

```
1  for i = |V| to 1 do
2      v ← V_O [i] ;
3      switch v in
4          case v ∈ L
5              ⟨P_V, #Sel⟩ ← FKBSelLeaf(v, P_V, #Sel, L_P) ;
6          end
7          case v ∈ A
8              ⟨P_V, #Sel, ξ⟩ ← FKBSelAnd(v, P_V, #Sel, ξ, ×, max) ;
9          end
10         case v ∈ O
11             ⟨P_V, #Sel, η⟩ ← FKBSelOr(v, P_V, #Sel, η, max) ;
12         end
13     end
14 end
15 return ⟨η, ξ, #Sel (1)⟩ ;
```

The first part of Alg. 6.1, as with the $k = 1$ algorithm, involves processing the sd-DNNF from the leaves to the root. This algorithm, Alg. 6.2, computes the k best selections using dynamic programming. At each node we compute and cache the k best selections rooted at that node, where a selection is summarized at each node based on its children's summaries. For **And** nodes, a selection is summarized by specifying, for each child, one of the child's selections. For **Or** nodes, a selection

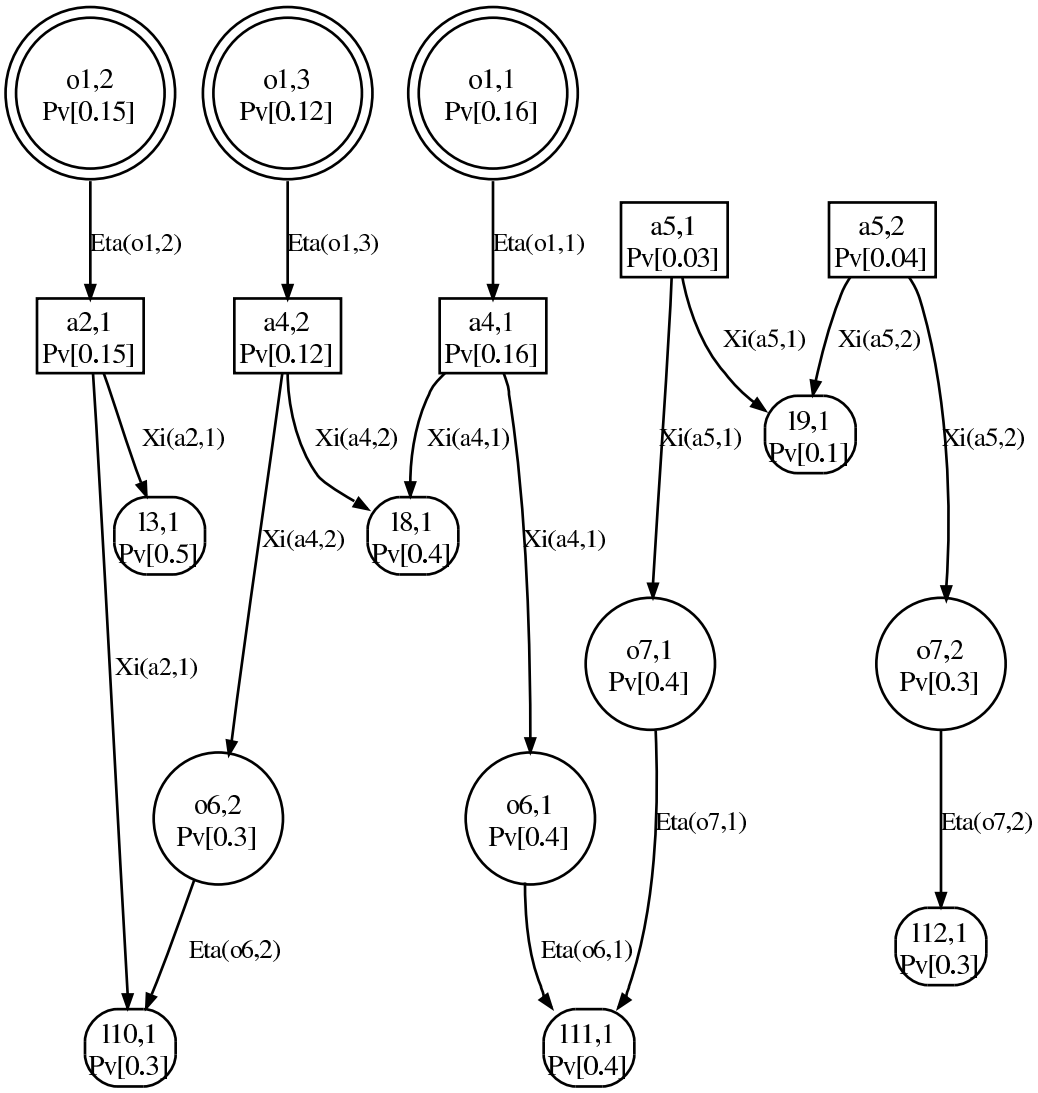

Figure 6-5: This is the modified graph of the A-B example once the FKBS algorithm, Alg. 6.2 has run.

is summarized by specifying a child and a selection for that child.

The algorithm has three update rules, one for each type of node: L, A, and O. These three rules are shown on Page 106. We have broken these three rules into three functions: Rule 1 is implemented in Alg. 6.3, Rule 2 is implemented in Alg. 6.6, and Rule 3 is implemented in Alg. 6.10.

Alg. 6.2, between lines 1-14, iterates over the nodes of the sd-DNNF, from the leaves to the root, invoking the appropriate rule. These rules update $\mathbf{P}_V$, #Sel, η, and ξ, as appropriate. $\mathbf{P}_V$ stores the value of each node's k best selections. For a leaf l, this always has one entry, equal to $\mathcal{L}_{\mathbf{P}}(l)$. For **And** and **Or** nodes, this stores the value of each selection, sorted from best to worst. Keeping this list sorted makes both **And** and **Or** node computations much more efficient. #Sel stores exactly how many selections are available at each node. This is between 1 and k. η records the k best child selections for **Or** nodes, with one child selection per **Or** node selection. ξ records the k best combinations of child selections for **And** nodes, with one entry per **And** node selection and child. We store all four variables with the node, eliminating the node index for efficient access. We pass ξ and η to GetKSolutionsFromSelections by passing the graph annotated with these variables.

This algorithm returns on Line 15, where it returns its selections collectively in ξ and η, along with the number of selections found at the root; this is exactly the number of corresponding solutions. In general, unless k is greater than the total number of solutions in the sd-DNNF, or unless the algorithm suppresses solutions with values less than a certain amount, the number of solutions found at the root is exactly k. In our motivating application of estimation, for example, we suppress solutions with 0 value (probability).

For the A-B Example, if $k = 3$, then Alg. 6.2, FindKBestSelections, adds the edges shown in Fig. 6-5 to the modified graph. We first illustrate what the algorithm does when it visits a leaf using the leaf l11 and then we focus on the edges added to o6 and a4. For the leaf case, Alg. 6.3 sets #Sel (l11) to 1, denoting that the leaf has one possible selection, and sets $\mathbf{P}_V(\text{l11}, 1)$ to $\mathcal{L}_{\mathbf{P}}(\text{l11}) = 0.4$, the value of this selection.

When Alg. 6.2 visits o6, the algorithm calls the sub-routine FKBSelOr, Alg. 6.10. This sub-routine sets #Sel (o6) $= 2$. For the first selection, Alg. 6.10 sets $\mathbf{P}_V(\text{o6}, 1) = 0.4$ and $\eta(\text{o6}, 1) =$

$\langle \text{l11}, 1\rangle$, where the modified child node $\langle \text{l11}, 1\rangle$ has value 0.4. For the second selection, Alg. 6.10 sets $\mathbf{P}_V(\text{o6}, 2) = 0.3$ and $\eta(\text{o6}, 2) = \langle \text{l10}, 1\rangle$.

When Alg. 6.2 visits a4, the algorithm calls the sub-routine FKBSelAnd, Alg. 6.6. This sub-routine sets $\#\text{Sel}(\text{a4}) = 2$. For the first selection, Alg. 6.6 sets $\mathbf{P}_V(\text{a4}, 1) = 0.16$, $\xi(\text{a4}, 1, \text{l8}) = 1$, and $\xi(\text{a4}, 1, \text{o6}) = 1$. For the second selection, Alg. 6.6 sets $\mathbf{P}_V(\text{a4}, 2) = 0.12$, $\xi(\text{a4}, 2, \text{l8}) = 1$, and $\xi(\text{a4}, 2, \text{o6}) = 2$. Thus we have $\langle \text{a4}, 1\rangle \rightarrow \langle \text{o6}, 1\rangle$ and $\langle \text{a4}, 2\rangle \rightarrow \langle \text{o6}, 2\rangle$ along with $\langle \text{a4}, 1\rangle \rightarrow \langle \text{l8}, 1\rangle$ and $\langle \text{a4}, 2\rangle \rightarrow \langle \text{l8}, 1\rangle$.

We now present Algorithms 6.3, 6.6, and 6.10 in Sections 6.3.1, 6.3.2, and 6.3.9, respectively.

6.3.1 Find-K-Best-Selections Leaf-case Algorithm

Algorithm 6.3: FKBSelLeaf(l, $\mathbf{P}_V$, #Sel, $\mathcal{L}_{\mathbf{P}}$)

1 $\mathbf{P}_V(l, 1) \leftarrow \mathcal{L}_{\mathbf{P}}(l)$;
2 $\#\text{Sel}(l) = 1$;
3 **return** $\langle \mathbf{P}_V, \#Sel\rangle$;

The leaf node case for finding the k best selections, Alg. 6.3, is nearly identical to that for Alg. A.2. We set $\mathbf{P}_V(l, 1) = \mathcal{L}_L(l)$ on Line 1 and record that we only have one selection on Line 2. This is illustrated on page 116.

Time and Space Analysis We store all variables indexed by an sd-DNNF node with the sd-DNNF node itself, so all look-up times are $O(1)$. Since a leaf always stores exactly one answer, a leaf need only have $O(1)$ space to store the two values. Thus, FKBSelLeaf requires $O(1)$ time and space.

6.3.2 Find-K-Best-Selections And-case Algorithm

The **And** node case for finding the k best selections, Alg. 6.6, requires finding the k best combinations of its children's selections. If this node a has $|E_a|$ children and each has k solutions, then there are $k^{|E_a|}$ combinations; however, we are only interested in k of them. Much less work is required to extract only k solutions, which we quantify momentarily.

	0.4	0.3	0.2	0.1
0.4	**0.16**	**0.12**	0.08	0.04
0.3	**0.12**	**0.09**	0.06	0.03
0.2	0.08	0.06	0.04	0.02
0.1	0.04	0.03	0.02	0.01

Table 6.1: This table illustrates the combinations of the children of a hypothetical **And** node with two children when $k = 4$. The two children have identical distributions of 0.4, 0.3, 0.2, and 0.1 for selections 1, 2, 3, and 4, respectively. The upper-left region circumscribes all combinations of the two children that could ever be part of the best 4 selections of the **And** node. The four bold values forming a square in the upper left are the 4 best selections for this example.

	0.4	0.3	0.2	0.1
0.4	**0.16**	**0.12**	0.08	0.04
0.3	**0.12**	0.09	-	-
0.25	**0.10**	-	-	-
0.05	0.04	-	-	-

Table 6.2: This table illustrates a second possible combination of the 4 best children, again in bold. We have omitted those entries that could never be optimal.

	0.5	0.2	0.2	0.1
0.35	**0.175**	0.07	0.07	0.035
0.3	**0.15**	0.06	-	-
0.2	**0.1**	-	-	-
0.15	**0.075**	-	-	-

Table 6.3: This table illustrates a third possible combination of the 4 best children, again in bold. We have omitted those entries that could never be optimal.

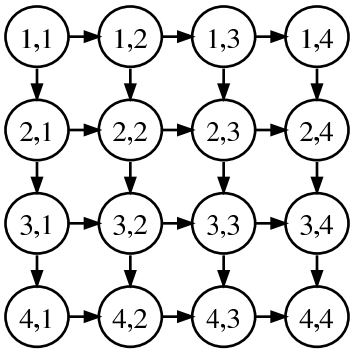

Figure 6-6: This figure shows which combinations of an **And** node's children is enabled in the case where $k = 4$ and the **And** node has two children.
A node in this graph is enabled if all of its parents have been included in the solution. Thus the root node in this graph, $1, 1$, is always enabled.

Lets start with an example. Let $k = 4$, $c = 2$, and $\mathbf{P}_V$'s entries 1 through 4 are 0.4, 0.3, 0.2, and 0.1, respectively, for both children. The combinations of these pair of children are illustrated in Table 6.1. The biggest combinations of the children are $\langle 1, 1\rangle$, $\langle 1, 2\rangle$, $\langle 2, 1\rangle$, and $\langle 2, 2\rangle$. The double-edge region defined around the upper left section of the matrix illustrates the region in which all k-best combinations reside. We illustrate two other combinations in Tables 6.2 and 6.3, which with their reflections, represent all k-best combinations of four.

The key to realize here is that, since our values are sorted by $\max$, and we only want the first k of them, we can start with the guaranteed best pair, the combination $\langle 1, 1\rangle$. We now show why this is the guaranteed best pair. The product of two numbers is monotonically increasing (unless one value is 0); when you increase either value of the product, the value of the product increases. Thus, the product of the two largest values is the largest value among all products. The next best product is a combination of the largest value of one of the two children and the second largest of the other child. Again, if we select the second best value for both children, the result is smaller than if we only decrease one of the values.

For a combination $\langle i, j\rangle$, the children of this combination are the combinations $\langle i + 1, j\rangle$ and $\langle i, j + 1\rangle$, subject to neither child index exceeding k. The parent/child relationship between combinations is illustrated for $k = 4$ in Fig. 6-6. The value of a combination $\langle i, j\rangle$ is always greater than

that of its children. Again, this trivially holds as one of the two values of the child is equal to one of this combination's values and the other child's value is less than this combination's other value.

As a corollary, a combination $\langle i, j \rangle$ need not be considered until all of its parents are considered. Since $\langle 1, 1 \rangle$ is the only node with no parents, this is the only possible maximal node, as we said above. We use this fact in Alg. 6.6 to pre-build a structure C_a to hold all possible combinations of k and then Alg. 6.6 need only consider among those combinations that have had all their parents selected in C_a.

To bound the number of combinations needed in C_a, we note that in order to consider the candidate at $\langle i, j \rangle$, all ancestors back to the first candidate $\langle 1, 1 \rangle$ must have already been accepted as part of the **And** node's selection. This means that there has been less than k combinations accepted. Since the ancestors of $\langle i, j \rangle$ form a filled in square and $\langle i, j \rangle$ may be the kth combination, $i * j \leq k$. Since all three values are positive, $j \leq \frac{k}{i}$. All the values are also integers, so $j \leq \lfloor \frac{k}{i} \rfloor$. We can count the total number of combinations by varying i from 1 to k, at which point the upper bound on j is the number of nodes allowed in the j dimension. Thus, the total number of possible candidates is:

$$\sum_{i=1}^{k} \left\lfloor \frac{k}{i} \right\rfloor$$

Each term of this equation is the floor of k times a term in the harmonic series[13]. The sum of the first k terms of the harmonic series is upper-bounded by $(\log k) + 1$, and thus the number of possible candidates is upper-bounded by $(k \log k) + k$ or $O(k \log k)$.

The algorithm in the prior work [49] approaches this problem in a less efficient manner for a fixed k values, but that scales to any value k. Specifically, the algorithm uses a queue of the current combinations being considered, but considers any combination $\langle i, j \rangle$ that is enabled along only one dimension. For combinations of the form $\langle 1, j \rangle$, a node is considered when the $\langle 1, j - 1 \rangle$ combination is accepted. For all other combinations $\langle i, j \rangle$, the combination is considered when the $\langle i - 1, j \rangle$ combination is accepted. Our approach, by contrast, will only consider nodes that have both parents accepted. The difference between these two algorithms is that combinations with one out of two nodes accepted will be in the queue for the algorithm proposed in Sachenbacher [49] and not in the queue for our algorithm. In the worst case, the algorithm of Sachenbacher [49] will insert

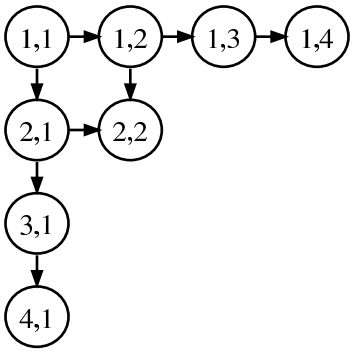

Figure 6-7: This figure shows the set of combinations that are part of the C_a structure when $k = 4$. A combination is enabled if both of its parents are accepted as part of the k best selections. $1, 1$ is always enabled.

$k - 2$ extra elements in the queue and increase the time and space complexity of this algorithm by about a factor of 2. In the best case both algorithms will insert the same number of elements. Note that our algorithm is only universally faster if a time stamp service is used to avoid needing to initializing the whole C_a structure each use at a cost of $k \log k$. The savings of this work are exclusively in terms of avoiding additional enqueue and dequeue costs for a larger queue.

To illustrate the relationship between the two algorithms, consider a simple example with $k = 3$ and two input selections with values $\{0.9, 0.1, 0.1\}$ and $\{0.4, 0.3, 0.3\}$. The three best combinations are $\langle 1, 1\rangle$, $\langle 1, 2\rangle$, and $\langle 1, 3\rangle$. The algorithm [49] will additionally consider the combinations $\langle 2, 1\rangle$, $\langle 2, 2\rangle$, while our algorithm will only consider the combination $\langle 2, 1\rangle$. The best case for algorithm [49] is when the input selections are reversed.

Fig. 6-7 shows the combinations of C_a trimmed down from Fig. 6-6. To make our algorithm efficient, combinations of C_a are all indexed to allow for constant look-up. Specifically, a combination c_a in C_a is a tuple $\langle i,\ j_1,\ j_2,\ i_1,\ i_2,\ \#_P,\ \#_E\rangle$. The combination is located at $C_a[i]$. The pair $\langle j_1, j_2\rangle$ specifies the selection for the two children. The two values i_1 and i_2 are indices in C_a, referring to the two children of c_a, $\langle j_1 + 1, j_2\rangle$ and $\langle j_1, j_2 + 1\rangle$. These can have the special value $\perp$ if the combination has 0 or 1 child. $\#_P$ is the number of parents of this entry, between 0 and 2. $\#_E$ is set when we use C_a, and corresponds to the current number of un-accepted parents, where

a parent is accepted when its combination is the next best combination. When we reset C_a, we set $\#_E = \#_P$. Each time a parent is accepted, it decrements both of its children's $\#_E$ value. When $\#_E$ reaches 0, the combination is enabled and can be added to the queue of enabled combinations. We require that the combination $\langle j_1, j_2 \rangle = \langle 1, 1 \rangle$ have a known index so we can start Alg. 6.6. Our algorithm for indexing C_a currently indexes the $\langle 1, 1 \rangle$ combination as the last index of C_a.

6.3.3 Construct-Combinations Algorithm

Algorithm 6.4: ConstructCombinations(C_a, k, j_1, j_2)

```
1  if j1 * j2 > k then
2      return ⟨Ca, ⊥⟩ ;
3  end
4  if ⟨j1, j2⟩ is in Ca then
5      i ← Index of ⟨j1, j2⟩ in Ca ;
6      return ⟨Ca, i⟩ ;
7  end
8  ⟨Ca, i1⟩ ← ConstructCombinations(Ca, k, j1 + 1, j2) ;
9  ⟨Ca, i2⟩ ← ConstructCombinations(Ca, k, j1, j2 + 1) ;
10 #P ← 0 ;
11 if j1 > 1 then
12     #P ← #P + 1 ;
13 end
14 if j2 > 1 then
15     #P ← #P + 1 ;
16 end
17 i ← |Ca| + 1 ;
18 Ca[i] ← ⟨i, j1, j2, i1, i2, #P, #P⟩ ;
19 return ⟨Ca, i⟩ ;
```

We show the code used to construct and initialize C_a in Alg. 6.4 and 6.5, respectively. C_a is used to make the **And** case of finding the k best selections more efficient. Alg. 6.4 is assumed to run prior to the other algorithms of this chapter, Alg. 6.1, as the data of C_a with the exception of $\#_E$ is constant for a constant k. The recursive Alg. 6.4 is called as ConstructCombinations(C_a, k, 1, 1), with an empty C_a, and returns a constructed C_a along with the location i of entry $c_a = \langle i, 1, 1, *, *, 0, 0 \rangle$. In general, this algorithm returns the updated C_a and the index of the entry with

the combination $\langle j_1, j_2 \rangle$, or $\perp$ if it isn't one of the possible combinations. This is done recursively, where we return the index of a entry if it has already been inserted into C_a and we insert a new entry into C_a, otherwise. An entry is inserted after all of its children have been inserted. If we assume $k \log k$ entries are generated, this algorithm looks through this list once for each edge in the graph to be sure the entry has not yet been created, where there are two edges per entry. Otherwise, it only performs $O(1)$ steps in order to compute the elements of the new entry.

Line 1 is the base case of Alg. 6.4. We return the non-entry index $\perp$ if there is no way for both parents of this entry to be accepted at the same time. Line 4 makes sure that we do not insert a combination more than once. If the combination already exists, we return the index of the combination's entry. As stated above, this is an $O(k \log k)$ operation in general[1]. Lines 8 and 9 recursively look-up or construct the two children of this combination.

Lines 10-16 set the number of parents of the entry. This is easily computed as most entries have two parents. An entry that has a value of 1 for one of its two combination values has only 1 parent, as the parent in the value-of-1 direction has a 0 value, and 0 is an invalid value (our values start at 1). The combination $\langle 1, 1 \rangle$ is the only combination where both values of the combination are 1, and so it has 0 parents.

Line 17 computes the index of this new entry. We insert this entry at the end of C_a. We then add our new node on Line 18 and return it on Line 19.

For example, consider the example C_a shown in Fig. 6-7, for $k = 4$. As stated above, we start by calling ConstructCombinations(C_a, 4, 1, 1), where C_a is initially empty. This pair $\langle 1, 1 \rangle$ does not exceed k nor is it already in C_a, so the algorithm recurses on Line 8 by calling ConstructCombinations(C_a, 4, 2, 1). The algorithm continues to recurse on the same line into ConstructCombinations(C_a, 4, 3, 1), ConstructCombinations(C_a, 4, 4, 1), and finally ConstructCombinations(C_a, 4, 5, 1). The algorithm then observes that $5 * 1 > 4$ on Line 1 and thus returns our cache C_a and $\perp$. Thus, for ConstructCombinations(C_a, 4, 4, 1), $i_1 = \perp$. This step in the recursion then recurses on Line 9, calling ConstructCombinations(C_a, 4, 4, 2), and this also returns $\perp$, so $i_2 = \perp$. The node

[1]We could speed this up to $O(\log(k \log k))$ if we add an explicit indexing map or $O(1)$ if we used an appropriate hashing function of $\langle j_1, j_2 \rangle$. These optimizations are ignored in this thesis because this is a pre-processing step and is fast enough for all our values of k.

$\langle 4, 1\rangle$ has only one parent, since the j_2 edge is along the top edge, so $\#_P$ is set to 1. This is the first entry, so i is set to one on Line 17 and then the entry is added to the end of C_a, at entry i, with the value $\langle 1, \langle 4, 1\rangle, \perp, \perp, 1, 1\rangle$. This routine then returns to ConstructCombinations(C_a, 4, 3, 1) that the $\langle 4, 1\rangle$ node is at index one. The node $\langle 3, 1\rangle$ is then added to C_a at index two with the value $\langle 2, \langle 3, 1\rangle, 1, \perp, 1, 1\rangle$. The algorithm continues and builds this full set of entries in C_a:

1. $\langle 1, \langle 4, 1\rangle, \perp, \perp, 1, 1\rangle$
2. $\langle 2, \langle 3, 1\rangle, 1, \perp, 1, 1\rangle$
3. $\langle 3, \langle 2, 2\rangle, \perp, \perp, 2, 2\rangle$
4. $\langle 4, \langle 2, 1\rangle, 2, 3, 1, 1\rangle$
5. $\langle 5, \langle 1, 4\rangle, \perp, \perp, 1, 1\rangle$
6. $\langle 6, \langle 1, 3\rangle, \perp, 5, 1, 1\rangle$
7. $\langle 7, \langle 1, 2\rangle, 3, 6, 1, 1\rangle$
8. $\langle 8, \langle 1, 1\rangle, 4, 7, 0, 0\rangle$

And it returns this list and 8, the index of the root node $\langle 1, 1\rangle$.

Time and Space Analysis We intend ConstructCombinations to be an off-line algorithm, as it generates a constant structure that depends only on k, thus the time it takes to generate C_a is not included in the time complexity of the other algorithms of this chapter, just in the space complexity of these other algorithms. Lines 1 and 10-19 are all $O(1)$ operations: reading or setting a field, or appending to the end of a vector. We stated above that Line 4 is currently just a linear search through a vector of length $O(k \log k)$, which is thus an $O(k \log k)$ operation. We could create an index that maps $\langle j_1, j_2\rangle$ to an index in C_a or $\perp$ to reduce this search cost, using a map or hash map. Lines 8 and 9 are recursive calls. We construct at most $O(k \log k)$ entries and we only recurse twice for constructed entries, so ConstructCombinations is called at most twice as many times as there are entries, still $O(k \log k)$. We only run Line 4 for constructible entries. Since an entry has at most

two parents, Line 4 is run at most twice per entry constructed, an $O(n^2)$ step for the algorithm, where n is the number of nodes generated and so $n = k \log k$. Thus, the overall complexity of ConstructCombinations is $O(k^2 \log^2 k)$ time. We generate only enough space to hold the entries we want, so the space required is the space of C_a, which we explained above is bounded by $O(k \log k)$.

6.3.4 Reset-Combinations Algorithm

Algorithm 6.5: ResetCombinations(C_a)

1 **foreach** $\langle i, j_1, j_2, i_1, i_2, \#_P, \#_E \rangle = C_a[i]$ **do**
2 $\quad C_a[i] \leftarrow \langle i, j_1, j_2, i_1, i_2, \#_P, \#_P \rangle$;
3 **end**
4 **return** C_a ;

The Alg 6.5 is just responsible for setting $\#_E = \#_P$ for each entry in C_a, specifically on Line 2. This is done iteratively. Thus the complexity of this algorithm is $O(|C_a|)$ time where $|C_a|$ is $O(k \log k)$.

6.3.5 The Find-K-Best-Selections And-case Algorithm Implementation

Next consider Alg. 6.6. This algorithm pair-wise combines all of the children's selections into this **And** node a's best k selections. The algorithm starts out by inheriting the selections of one of its children. Then, for all of the other children, it computes the best k selections from the combination of a's current selections and the next child's selections. Once all children have been combined, the algorithm's current k best selections are the actual k best selections and the algorithm is done.

In Alg. 6.6, Line 1 starts out by getting some out-going edge of the **And** node a, for child n. Line 2 inherits the best selections of the child n as a's best selections, noting which child these selections came from, using Alg. 6.7. The variable $\#_a$ stores the current number of selections, between 1 and k. The variable $\mathbf{P}_a$ is a local version of $\mathbf{P}_V$ specific to a.

The variable β_ξ is used to compute the entries for ξ, the set of k best selections for the **And** node. β_ξ is an acyclic graph that captures the best combinations of a child n of a with all other children that have already been combined. β_ξ is a function $V \times \{1, \ldots, k\} \rightarrow (V \cup \{\perp\}) \times \{1, \ldots, k\} \times$

Algorithm 6.6: FKBSelAnd(a, $\mathbf{P}_V$, #Sel, ξ, $\times$, max)

```
1  e ← some ⟨a, n_Prev⟩ ∈ E ;
2  ⟨#_a, P_a, β_ξ⟩ ← InheritFirstChild(n_Prev, #Sel, P_V) ;
3  foreach ⟨a, n⟩ ∈ E \ {e} do
4      ⟨#_a, P_a, β_ξ, n_Prev⟩ ← MergePair(β_ξ, n_Prev, n, P_a, P_V, #_a, #Sel (n)) ;
5  end
6  for i = 1 to #_a do
7      P_V (a, i) ← P_a (i) ;
8      ⟨n, j_1⟩ ← ⟨n_Prev, i⟩ ;
9      while n ≠ ⊥ do
10         ⟨n', j'_1, j_2⟩ ← β_ξ (n, j_1) ;
11         ξ (a, i, n) ← j_2 ;
12         ⟨n, j_1⟩ ← ⟨n', j'_1⟩ ;
13     end
14 end
15 #Sel (a) = #_a ;
16 return ⟨P_V, #Sel, ξ⟩ ;
```

$\{1, \ldots, k\}$, where V are the children of a. For a particular entry $\beta_\xi(v, i) = \langle v_2, i_2, j\rangle$, the entry means that the child selection $\langle v, j\rangle$ is part of the ith best selection of a, $\langle a, i\rangle$, and that the entry at $\beta_\xi(v_2, i_2)$ is also part of the ith best selection of a. A leaf of this graph an entry of the form $\langle \perp, 1, j\rangle$, for some j.

Consider an example where $k = 3$ and there are three children, $|E_v| = 3$. In this example, the children are n_1, n_2, and n_3 and all of them have three selections. These selections have values such that the three best combinations of n_1 and n_2 are $\rho_1 = (\langle n_1, 1\rangle, \langle n_2, 1\rangle)$, $\rho_2 = (\langle n_1, 2\rangle, \langle n_2, 1\rangle)$, and $\rho_3 = (\langle n_1, 1\rangle, \langle n_2, 2\rangle)$. Given these three best combinations of n_1 and n_2, the values are such that the three best combinations of these combinations and n_3 are $(\rho_1, \langle n_3, 1\rangle)$, $(\rho_1, \langle n_3, 2\rangle)$, and $(\rho_2, \langle n_3, 1\rangle)$, where the index of ρ is the ith best combination of n_1 and n_2. We can rewrite these combinations without indices as $((\langle n_1, 1\rangle, \langle n_2, 1\rangle), \langle n_3, 1\rangle)$, $((\langle n_1, 1\rangle, \langle n_2, 1\rangle), \langle n_3, 2\rangle)$, and $((\langle n_1, 2\rangle, \langle n_2, 1\rangle), \langle n_3, 1\rangle)$, respectively. This example is depicted in Fig. 6-8. The three best combinations of a can be read from Fig. 6-8 by looking at the three sequences that start at the three top nodes $\langle n_3, 1\rangle$, $\langle n_3, 2\rangle$, and $\langle n_3, 3\rangle$, respectively. Reading off the three sequences, in reverse –

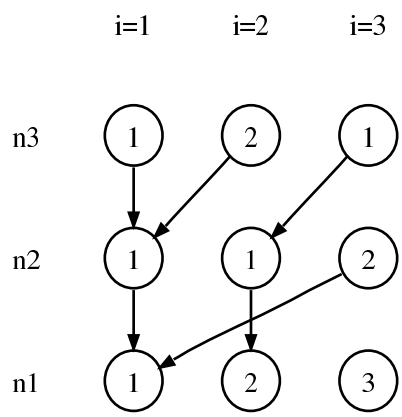

Figure 6-8: This is an example of a possible configuration of β_ξ for an example that assumes $k = 3$ and $|E_v| = 3$. The labels on the nodes are the value j for the entry $\beta_\xi(n, i) = \langle n_2, i_2, j \rangle$. As i increases, the value of the entry decreases. Each row represents the best k combinations of that node n_i including all $n_1 \ldots n_{i-1}$ below it. The arrow points to the n_{i-1} combination that is included in the n_i row's i^{th} best combination. The top row represents the k best combinations for all of the children of the **And** node, and thus the **And** node itself as well.
For example, the entry $(n_2, i = 2)$ is the second best combination considering n_1 and n_2. It includes the second best combination of n_1, which is the second best value for n_1, and thus $(n_1, 2)$ is labeled with 2. $(n_2, 2)$ is labeled with 1, which means that this second best combination considering n_1 and n_2 includes the best value for n_2. This second best combination considering n_1 and n_2 is the third best overall combination considering all three children, and thus $(n_3, i = 3)$ points to $(n_2, i = 2)$.

from n_1 to n_3, we get the same three best combinations $\langle 1,1,1\rangle$, $\langle 1,1,2\rangle$, and $\langle 2,1,1\rangle$. The nine entries of β_ξ that correspond to Fig. 6-8 are:

$$\begin{array}{lll}
\beta_\xi(n_3,1) = \langle n_2,1,1\rangle & \beta_\xi(n_3,2) = \langle n_2,1,2\rangle & \beta_\xi(n_3,3) = \langle n_2,2,1\rangle \\
\beta_\xi(n_2,1) = \langle n_1,1,1\rangle & \beta_\xi(n_2,2) = \langle n_1,2,1\rangle & \beta_\xi(n_2,3) = \langle n_1,1,2\rangle \\
\beta_\xi(n_1,1) = \langle \perp,1,1\rangle & \beta_\xi(n_1,2) = \langle \perp,1,2\rangle & \beta_\xi(n_1,3) = \langle \perp,1,3\rangle
\end{array}$$

Alg. 6.6 constructs β_ξ one row at a time, where Alg. 6.7 constructs the bottom row of β_ξ, and each subsequent row is added by Alg. 6.8. After calling Alg. 6.7 on Line 2, β_ξ contains $\#_a$ entries, where the ith entry is $\beta_\xi(n_{\text{Prev}}, i) = \langle \perp, 1, i\rangle$.

Lines 3-5 loop over all the remaining children of a, taking the k best combinations of a's current k best combinations and the child's k best combinations, inserting another row in β_ξ. This loop utilizes the function MergePair, Alg. 6.8. Finally, lines 6-15 copy the local versions of these variables over to the final version. Lines 8-13 copies the ith best combination of a from β_ξ into $\xi(a,i,*)$. Line 8 sets our current node in β_ξ to the root node of the ith best combination in β_ξ; the root is the ith position of the top row. Lines 9-13 loop from the root node in β_ξ to the leaf, where at the end $n = \perp$. For each node in β_ξ visited, the algorithm grabs the node's data on Line 10. This data specifies the next node in the sequence as well as a modified graph's node $\langle n, j_2\rangle$ (the j_2^{th} selection) that belongs to the ith combination of a. The algorithm connects this modified node to $\langle a, i\rangle$ on Line 11 and then the loop moves on to the next β_ξ node in the sequence on Line 12.

Consider again the A-B Example, within Alg. 6.6, FKBSelAnd. We assume that the first edge of the **And** node a4 that we choose points towards l8. With this assumption, Line 2 initializes our three local **And** node variables to: $\#_a = 1$, $\mathbf{P}_a(1) = 0.4$, and $\beta_\xi(\text{l8}, 1) = \langle \perp, 1, 1\rangle$. The function MergePair, Alg. 6.8, is then run on the only pairing, between l8 and o6. This pairing sets $\beta_\xi(\text{o6}, 1) = \langle \text{l8}, 1, 1\rangle$ and $\beta_\xi(\text{o6}, 2) = \langle \text{l8}, 1, 2\rangle$. These three entries summarize two combinations, $\langle 1,1\rangle$ and $\langle 1,2\rangle$, with two edges each, thus describing the four edges presented just previously for ξ.

Algorithm 6.7: InheritFirstChild(n, #Sel, $\mathbf{P}_V$)

```
1 #_a = #Sel (n) ;
2 for i = 1 to #_a do
3     P_a (i) = P_V (n, i) ;
4     β_ξ (n, i) ← ⟨⊥, 1, i⟩ ;
5 end
6 return ⟨#_a, P_a, β_ξ⟩ ;
```

6.3.6 Inherit-First-Child Algorithm

The first subroutine of Alg. 6.6 is InheritFirstChild, Alg. 6.7. This algorithm is responsible for initializing the bottom row of β_ξ along with the value of each column in this bottom row in $\mathbf{P}_a$ from a child's best selections. The **And** node inherits the selections of the child node, so Line 1 sets $\#_a$ to the number of selections of the child. The values are the same and in the same order, so these can also be copied. These initialized values are returned on Line 6.

Time and Space Complexity This algorithm performs an $O(1)$ step on Line 1, copying the number of selections of a's first child. We then copy up to k values on lines 3 and 4. Both take $O(1)$ time. Thus, the time complexity of this algorithm is $O(k)$. The space required is dominated by β_ξ, requiring $O(k|E_a|)$ space, though this space is not specific to InheritFirstChild, as it is space returned to the calling function FKBSelAnd, Alg. 6.6.

6.3.7 Merge-Pair Algorithm

The other subroutine of Alg. 6.6 is MergePair, Alg. 6.8. This algorithm is responsible for computing the next row of β_ξ based on the previous row for n_{Prev} in combination with the next child n. The variable β_ξ summarizes the combinations of the processed children. We use our combination structure C_a from Alg. 6.4 and 6.5 to decide which combinations are available as we select our k best combinations. MergePair extracts up to k values and stores them in β_ξ and $\mathbf{P}'_a$. It is assumed that these two value vectors, $\mathbf{P}_a$ and $\mathbf{P}'_a$, are swapped between pairings, so that the value vector is only copied once, when it is copied to $\mathbf{P}_V$. This algorithm uses a priority queue of possible candidate combinations to efficiently insert combinations and extract the best combination. Combinations are

Algorithm 6.8: MergePair(β_ξ, n_{Prev}, n, $\mathbf{P}_a$, $\mathbf{P}_V$, $\#_a$, $\#_n$)

1 Let C_a be the pre-constructed combinations structure and $\#_{1,1}$ be the index of the combination $\langle 1, 1\rangle$;
2 $C_a \leftarrow$ ResetCombinations(C_a) ;
3 $\#'_a \leftarrow 0$;
4 $p \leftarrow \mathbf{P}_a(1) \times \mathbf{P}_V(n, 1)$;
5 Insert $\langle p, \#_{1,1}\rangle$ into Q ;
6 **while** $\#'_a < k$ **and** $|Q| > 0$ **do**
7 $\quad\langle p, i\rangle \leftarrow$ Remove best element of Q ordered by p using $\max$;
8 $\quad\langle i, j_1, j_2, i_1, i_2, \#_P, \#_E\rangle \leftarrow C_a[i]$;
9 $\quad\#'_a \leftarrow \#'_a + 1$;
10 $\quad\mathbf{P}'_a(\#'_a) = p$;
11 $\quad\beta_\xi(n, \#'_a) \leftarrow \langle n_{\text{Prev}}, j_1, j_2\rangle$;
12 $\quad\langle Q, C_a\rangle \leftarrow$ InsSucc(Q, C_a, $\mathbf{P}_a$, $\mathbf{P}_V$, n, $\times$, $\max$, $\#_a$, $\#_n$, i_1) ;
13 $\quad\langle Q, C_a\rangle \leftarrow$ InsSucc(Q, C_a, $\mathbf{P}_a$, $\mathbf{P}_V$, n, $\times$, $\max$, $\#_a$, $\#_n$, i_2) ;
14 **end**
15 **return** $\langle \#'_a, \mathbf{P}'_a, \beta_\xi, n\rangle$;

sorted by their value using $\max$.

The algorithm starts out on Line 2 by setting $\#_E = \#_P$ for all combinations in C_a. This marks each combination as initially having none of its parents accepted. We initialize the number of columns in the new row of β_ξ to 0 on Line 3. The best combination is always $\langle 1, 1\rangle$, and we compute the value of this combination on Line 4. We then insert this best value into the queue Q on Line 5. We Are now ready to extract the best k combinations between the **And** node's current best combinations and the new node's best selections. Lines 6-14 are responsible for getting the next best combination, recording it, and the adding that combinations successors to Q, as appropriate. Successors, here, are defined by the relationship stored in C_a.

Within this loop, Line 7 finds the next best combination, ordered with $\max$. Since a candidate is only inserted in the queue once all of its parents have already been added to $\mathbf{P}'_a$ and β_ξ, the maximal node in Q is the next most maximal combination of a. Line 8 looks up the combination associated with the index we got from Q. This gives us the combo $\langle j_1, j_2\rangle$ and up to two successors i_1 and i_2. Line 9 increments our number of accepted combinations by one as we just got a new one off the queue. Line 10 sets the value of our next accepted combination to p, the value we computed for

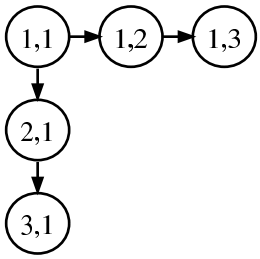

Figure 6-9: This figure shows the set of combinations that are part of the C_a structure when $k = 3$. A combination is enabled if both of its parents are accepted as part of the k best selections. $1, 1$ is always enabled.

sorting in Q. Line 11 points our next combination to the previous combination starting at $\langle n_{\text{Prev}}, j_1 \rangle$ and adds an entry for the child n based on its selection of j_2. Lines 12 and 13 call InsSucc on the first and second possible successors, respectively. InsSucc updates C_a by recording that one more parent of i_1 and i_2 has been accepted. If all of the child's parents have been accepted, InsSucc computes the value of the combination of the child and add it to the queue. Once the loop is done getting up to k combinations, the function returns the new local variables on 15.

For the A-B Example, MergePair, Alg. 6.8, requires that we have already constructed C_a for $k = 3$. This C_a is shown in Fig. 6-9. The entries $\langle i, j_1, j_2, i_1, i_2, \#_P, \#_E \rangle$ of C_a are:

- $C_a[1] = \langle 1, 3, 1, \perp, \perp, 1, 1 \rangle$
- $C_a[2] = \langle 2, 2, 1, 1, \perp, 1, 1 \rangle$
- $C_a[3] = \langle 3, 1, 3, \perp, \perp, 1, 1 \rangle$
- $C_a[4] = \langle 4, 1, 2, \perp, 3, 1, 1 \rangle$
- $C_a[5] = \langle 5, 1, 1, 2, 4, 0, 0 \rangle$

The first step of Alg. 6.8 sets the number of remaining parents $\#_E$ equal to the number of actual parents $\#_P$ for each entry by calling ResetCombinations Alg. 6.5. The entries listed above already have the two values equal. MergePair then sets our new number of solutions $\#'_a = 0$ and computes

the value of the first combination $\langle 1, 1\rangle$. The value of the first combination is $0.4 \times 0.4 = 0.16$. Line 5 inserts the entry $\langle 0.16, 5\rangle$ into Q, where 5 is the index of the $\langle 1, 1\rangle$ combination in C_a.

Alg. 6.8 then loops over lines 6-14. In the first iteration, the only element in Q is removed, the entry $\langle 0.16, 5\rangle$. The loop records that $\#'_a = 1$, sets $\mathbf{P}'_a(1) = 0.16$, and sets $\beta_\xi(\text{o6}, 1) = \langle \text{l8}, 1, 1\rangle$.

Line 12 then calls InsSucc, Alg. 6.9, for the combination $\langle 2, 1\rangle$, but this is not a valid combination as l8 does not have 2 selections, so InsSucc does nothing. Line 13 then calls InsSucc for the combination $\langle 1, 2\rangle$ and this both exists and is now enabled, so InsSucc computes the value of this combination $0.4 \times 0.3 = 0.12$ and inserts $\langle 0.12, 4\rangle$ into Q.

In the second iteration of Alg. 6.8, the entry $\langle 0.12, 4\rangle$ is dequeued from Q. The iteration records that $\#'_a = 2$, sets $\mathbf{P}'_a(2) = 0.12$, and sets $\beta_\xi(\text{o6}, 2) = \langle \text{l8}, 1, 2\rangle$. Alg. 6.8 then calls InsSucc on $i_1 = \perp$ in $C_a[4]$, so InsSucc immediately returns. Alg. 6.8 then calls InsSucc on $i_2 = 3$, which has the combination $\langle 1, 3\rangle$. Since $3 >$ #Sel (o6), InsSucc also immediately returns. The queue Q is then empty, with only 2 selections, and the algorithm returns with just these two selections.

Figures 6-10 and 6-10 illustrate the modified graph of the A-B example with $k = 2$, and $k = 1$, respectively. These figures show how the modified nodes in Fig. 6-5 are eliminated as the number of selections we seek is reduced.

6.3.8 Insert-Successor Algorithm

The last subroutine used by the **And** node case is the InsSucc algorithm. As was just stated, this routine updates the value of $\#_E$, the number of un-accepted parents, of the entry i in C_a. This involves subtracting one, as this function is called whenever a parent of this entry has been accepted. If $\#_E$ becomes zero, then the entry's combination is possibly the next best combination and thus becomes enabled and we insert it in the queue. This requires first computing the value of the combination.

There are two special cases for this routine. First, i may be equal to $\perp$, in which case this isn't actually referring to an entry and there isn't anything to do. Recall that this means that it was not possible for this child of the parent to have ever been enabled, so this child reference was set to $\perp$. The other case is that one or both of the nodes a and n may not have a full k selections. This matters

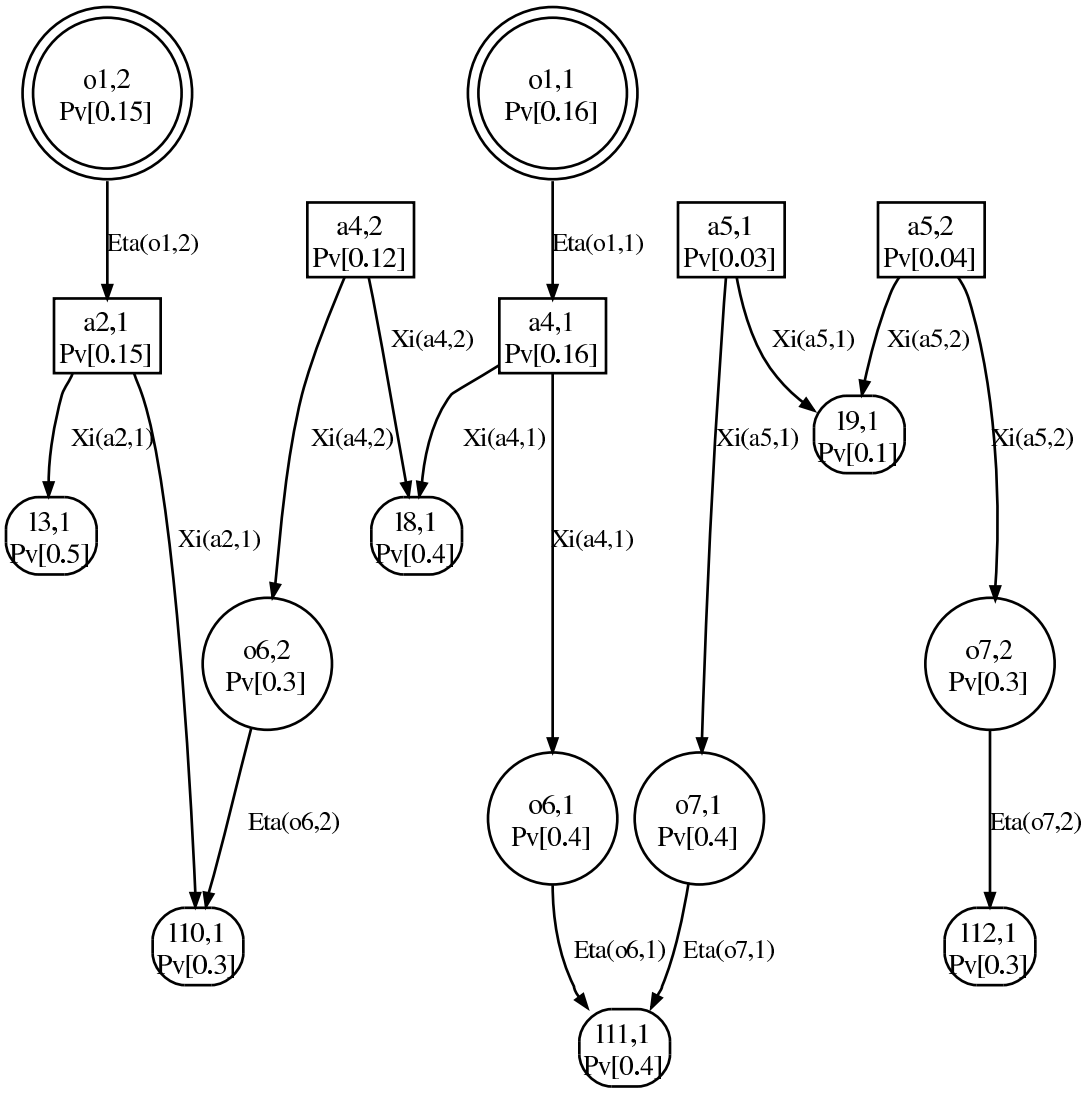

Figure 6-10: This figure shows how the modified nodes and edges change when $k = 2$ as opposed to $k = 3$.

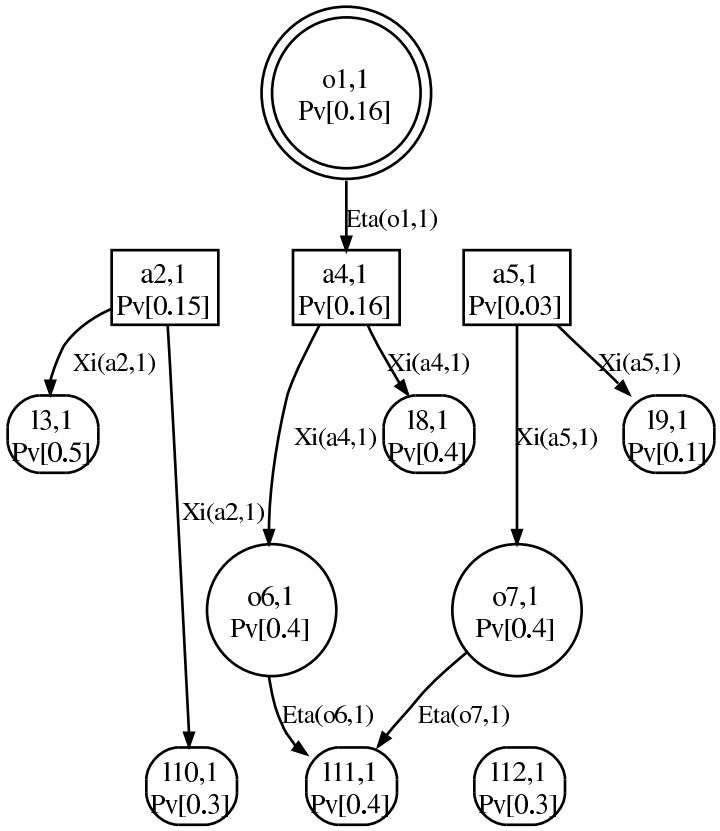

Figure 6-11: This figure shows how the modified nodes and edges change when $k = 1$ as opposed to $k = 2$.

Algorithm 6.9: InsSucc(Q, C_a, $\mathbf{P}_a$, $\mathbf{P}_V$, n, $\times$, $\max$, $\max_1$, $\max_2$, i)

```
1  if i = ⊥ then
2      return ⟨Q, C_a⟩ ;
3  end
4  ⟨i, j_1, j_2, i_1, i_2, #_P, #_E⟩ ← C_a[i] ;
5  if j_1 > max_1 or j_2 > max_2 then
6      return ⟨Q, C_a⟩ ;
7  end
8  #_E ← #_E − 1 ;
9  C_a[i] ← ⟨i, j_1, j_2, i_1, i_2, #_P, #_E⟩ ;
10 if #_E = 0 then
11     p ← P_a(j_1) × P_V(n, j_2) ;
12     Insert ⟨p, i⟩ into Q ordered by p using max ;
13 end
14 return ⟨Q, C_a⟩ ;
```

because we cannot compute the value of a combination if one of the values of the combination exceeds the number of selections of the corresponding node. We thus prune combinations that exceed our actual number of selections.

The algorithm starts out on Line 1 by returning if i is $\perp$. Line 4 gets the entry in C_a corresponding to i so we can update $\#_E$. Before updating i, the algorithm returns if the combination of i exceeds the number of selections of either sd-DNNF node on Line 5. Lines 8 and 9 update $\#_E$ in C_a. Lines 10-13 add the combination to Q if $\#_E$ is 0, which is to say if the combination is enabled. Line 11 computes the value of the combination, while Line 12 adds the combination to Q.

Runtime Analysis We start with the InsSucc routine and work up to the Find-K-Best-Selections, And-case algorithm. Every time the InsSucc routine is called, it looks up an entry in C_a, decrementing the value $\#_E$. Lines 1-9 are all $O(1)$, as we assume we update in-place and that we access directly by index. Lines 11 and 12 are only run once per combination inserted into the queue. Line 11 applies $\times$ once, an $O(1)$ operation. Line 12's complexity depends on the size of the queue. For a queue of length $|Q|$, Line 12 has a time complexity of $O(\log |Q|)$. This complexity arises from $O(\log |Q|)$ applications of $\max$ to determine where in the heap the combination belongs. We show that $|Q| \leq \left\lfloor \sqrt{2k} \right\rfloor + 1$, so Line 12's complexity is $O(\log k)$. Thus, if the candidate is not added to

the queue, InsSucc has $O(1)$ time complexity, and if it is added to the queue, InsSucc has $O(\log k)$ complexity. InsSucc requires $O(1)$ space for everything but Q. The queue, as we said, is of size $O(\sqrt{k})$.

We now justify the claim that the queue is never larger than $\left\lfloor \sqrt{2k} \right\rfloor + 1$. First, note that the algorithm accepts at most k combinations before terminating. Also note that we only add a combination $\langle i, j \rangle$ to the queue if all their parents have been accepted, and they have only been accepted if all their parents have been accepted, etc. For $\langle i, j \rangle$ to be added to the queue, $(i * j) - 1 \leq k$ combinations have been accepted. Additionally, note that at most one combination per row and per column is enabled and thus in the queue. Lets assume without loss of generality that there are two in the same row i. Let the column of the two combinations be j_1 and j_2, such that $j_1 < j_2$. The combination $\langle i, j_1 \rangle$ is an ancestor of $\langle i, j_2 \rangle$, and both are only enabled. This violates our constraint that a combination be accepted prior to any of its children being enabled, and thus any of their descendants being enabled. So there is at most one enabled combination per row and column. The configuration with k combinations accepted that has the maximal number of combinations enabled has one enabled combination per row and per column, lets say w rows and h columns. This is maximal for a fixed area k as inserting just a row or column anywhere increases the number of combinations accepted while not changing the number of enabled combinations. To compensate for the extra area added, we need to remove a column or row, which in turn reduce the number of enabled combinations. This maximal form is square, so $w = h$, and forms a triangle, which has area $\frac{1}{2}w^2$. The area of the triangle is equal to the number of accepted combinations, so $\frac{1}{2}w^2 \leq k$. Solving for w, we get $w \leq \sqrt{2k}$. Since w is an integer, a tighter bound is $w \leq \left\lfloor \sqrt{2k} \right\rfloor$. If we have at most one enabled combination per column, at most one enabled combination to the right of the right most accepted combination, and w columns of accepted combinations, then we can have at most $w + 1$ enabled combinations, or $\left\lfloor \sqrt{2k} \right\rfloor + 1$.

We showed previously that ResetCombinations has a complexity of $O(k \log k)$ time and $O(1)$ space, required to reset the $\#_E$ values of each entry of C_a. Thus, with the complexity of InsSucc, we can now analyze the complexity of the MergePair algorithm. The MergePair algorithm calls ResetCombinations once on Line 2, requiring $O(k \log k)$ time. Lines 3-5 are $O(1)$ time operations.

The queue Q is needed only for this sub-routine, starting at Line 5, and has a maximal space requirement of $O(\sqrt{k})$. Our loop from lines 6-14 runs at most k times. Within the loop, we dequeue an element from Q once on Line 7 with complexity $O(\log k)$. Lines 8-11 perform $O(1)$ operations, setting values. Lines 12 and 13 each call InsSucc. InsSucc inserts at most $k + \left\lfloor \sqrt{2k} \right\rfloor$ combinations into Q, with an $O(\log k)$ time complexity each time something is inserted into Q. The remaining times it is called it has an $O(1)$ time complexity. Thus, the loop excluding the InsSucc part has an $O(k \log k)$ time complexity. The InsSucc part has $O((k + \left\lfloor \sqrt{2k} \right\rfloor) \log k) = O(k \log k)$ time complexity. Thus, together the whole loop has time complexity $O(k \log k)$. The final Line 15 can have an $O(k)$ time complexity if the data $\mathbf{P}'_a$ is copied to $\mathbf{P}_a$ or $O(1)$ if the data is swapped. Given all three parts, the initial part, the loop, and the return (either version), the overall time complexity of MergePair is $O(k \log k)$. We now summarize the space required. The C_a uses $O(k \log k)$ space. The Q uses $O(\sqrt{k})$ space. Our local copy of $\mathbf{P}'_a$ uses $O(k)$ space. The β_ξ uses $O(|E_a|k)$ space, where each call to MergePair adds $O(k)$ data to β_ξ. Thus, the total space required for this step is $O(|E_a|k + k \log k)$.

We can now finally determine the complexity of the Find-K-Best-Selections, And-case (FKBSelAnd) algorithm. First recall that the complexity of InheritFirstChild was $O(k)$ time and $O(|E_a|k)$ space. We assume that out-going edges of a node are stored with the node, so Line 1 of Alg. 6.6 just involves selecting the first out-going edge, an $O(1)$ operation. Line 2 invokes InheritFirstChild once, returning our local variables $\langle \#_a, \mathbf{P}_a, \beta_\xi \rangle$. The local variables take $O(|E_a|k)$ space, where $|E_a|$ is the number of children of a, or equivalently the number of out-going edges. The algorithm then iterates over the remaining $|E_a| - 1$ edges on Line 3, and for each edge, it invokes MergePair. MergePair requires $O(k \log k)$ time and space, so the loop requires $O(|E_a|k \log k)$ time and, as we need not keep the previous local variable copies, only $O(|E_a|k + k \log k)$ space. Lines 6-15 copy the local variables to their final location on the node, an $O(|E_a|k)$ time operation. All together, this algorithm has an $O(|E_a|k \log k)$ time and an $O(|E_a|k + k \log k)$ space complexity.

Algorithm 6.10: FKBSelOr(o, $\mathbf{P}_V$, #Sel, η, $\max$)

```
1  E_o ← All edges matching ⟨o, n⟩ ∈ E, in any order ;
2  #_E ← |E_o| ;
3  for i = 1 to #_E do
4      ⟨o, n⟩ ← E_o[i] ;
5      p ← P_V (n, 1) ;
6      Insert ⟨p, n, 1, #Sel (n)⟩ in Q ordered by p using max ;
7  end
8  #_o ← 0 ;
9  while #_o < k and |Q| > 0 do
10     ⟨p, n, j, max_j⟩ ← Remove best element of Q ordered by p using max ;
11     #_o ← #_o + 1 ;
12     P_V (o, #_o) ← p ;
13     η (o, #_o) ← ⟨n, j⟩ ;
14     if j + 1 ≤ max_j then
15         p ← P_V (n, j + 1) ;
16         Insert ⟨p, n, j + 1, max_j⟩ in Q ordered by p using max ;
17     end
18 end
19 #Sel (o) = #_o ;
20 return ⟨P_V, #Sel, η⟩ ;
```

6.3.9 Find-K-Best-Selections Or-case Algorithm

The **Or** node case for finding the k best selections, Alg. 6.10, requires gathering the best k selections from all its children. This can be likened to performing the traditional merge step of a merge-sort[14], with two modifications. First, there are multiple lists, not just two. There is one list per child; since there are $|E_o|$ children, there are $|E_o|$ lists. Second, while each list may contain k elements, we are only interested in the first k merged elements, not all $k|E_o|$. We solve this problem by keeping a priority queue of the leading selections for each list. The algorithm repeatedly takes the best option from the queue and then adds to the queue the associated list's next best element, if any.

Alg. 6.10 starts out by getting a reference to all of its edges on Line 1. The algorithm records the number of edges on Line 2. Lines 3-7 setup our $|E_o|$ lists for merging, inserting each one into the priority queue Q. Since each child's selections are sorted, to get the next best answer for the

child the algorithm needs only look at the next selection for the child. Thus, the algorithm records in Q the value of this list's best option as well as its index so it can easily compute the next best index. The value of the best selection of each child is always the first one, $\mathbf{P}_V(n, 1)$, and this is looked-up on Line 5. The algorithm records 4 elements in an entry in Q on Line 6: $\langle p, n, j, \max_j \rangle$. p is the value of the entry, n is the child node of o, j is the selection number of node n that has value p, and $\max_j$ is the number of selections that n has. Only p, n, and j are necessary, as $\max_j$ can be looked-up based on n, but it is convenient to include $\max_j$. Line 8 sets the number of selections gathered to 0.

Lines 9-18 take the next best selection from among the current selections of each list from the Q and records it. If there is another selection after this current selection, it is re-inserted in Q. Line 10 gets our best element from the queue. Lines 11-13 record this next best entry into o's variables and increments the number of selections found. Lines 14-17 check to see if there is another selection for node n at position $j + 1$. If so, these lines get the value of this next selection and inserts an entry for this next selection in Q. Line 19 sets the number answers we found in this node's local variable and then the algorithm returns on Line 20.

Returning to the A-B Example, we demonstrate Alg. 6.10 on node o1. This algorithm identifies three selections: $\langle \text{a4}, 1 \rangle$, $\langle \text{a2}, 1 \rangle$, and $\langle \text{a4}, 2 \rangle$. These selections are assigned values in $\mathbf{P}_V$ and children in η. The node o1 has three children: a2, a4, and a5. It thus sets $\#_E = 3$ on Line 2. For all three of these children, the algorithm enqueues an entry of the form $\langle p, n, j, \max_j \rangle$. The children of o1 have the entries:

- a2: $\langle 0.15, \text{a2}, 1, 1 \rangle$
- a4: $\langle 0.16, \text{a4}, 1, 2 \rangle$
- a5: $\langle 0.04, \text{a5}, 1, 2 \rangle$

These are all inserted into Q on Line 6. Thus the queue has these three entries on Line 8, which sets our current number of selections, $\#_o$, equal to 0.

The first iteration of lines 9-18 starts out on Line 10 by removing the best entry, $\langle 0.16, \text{a4}, 1, 2 \rangle$ from Q. After this, Q contains only two entries: $\langle 0.15, \text{a2}, 1, 1 \rangle$ and $\langle 0.04, \text{a5}, 1, 2 \rangle$. Line 11 sets

the number of selections to 1. Line 12 sets $\mathbf{P}_V(\text{o1}, 1) = 0.16$. Line 13 records the modified child of o1 that had this value, namely $\langle \text{a4}, 1\rangle$. Thus, η now contains the edge $\langle \text{o1}, 1\rangle \rightarrow \langle \text{a4}, 1\rangle$.

Line 14 checks if $j + 1 \leq 2$, and it does. Thus, the algorithm inserts the next best value of a4 in the queue. The modified node $\langle \text{a4}, 2\rangle$ has the value $\mathbf{P}_V(\text{a4}, 2) = 0.12$, so the entry $\langle 0.12, \text{a4}, 2, 2\rangle$ is added to the Q on Line 16. The loop from lines 9-18 then starts again at the beginning.

The second iteration of the loop starts out by again removing the best entry from Q. In this case the best entry is $\langle 0.15, \text{a2}, 1, 1\rangle$. The loop sets $\#_o = 2$, $\mathbf{P}_V(\text{o1}, 2) = 0.15$, and $\eta(\text{o1}, 2) = \langle \text{a2}, 1\rangle$. This loop skips lines 14-17 because a2 does not have any more selections. The third and final iteration of the loop starts out by removing the next best entry of the Q, $\langle 0.12, \text{a4}, 2, 2\rangle$. This iteration sets $\#_o = 3$, which is also k, it also sets $\mathbf{P}_V(\text{o1}, 3) = 0.12$, and $\eta(\text{o1}, 3) = \langle \text{a4}, 2\rangle$. The loop then exits, setting the final number of selections of o1, #Sel (o1), to 3 on Line 19. At the end of the algorithm, Q still contains $\langle 0.04, \text{a5}, 1, 2\rangle$, which is never considered as it has a lower value, 0.04, than any of entries returned, of which the lowest value is 0.12.

Runtime Analysis The FKBSelOr algorithm has two loops, one that generates an initial set of candidates among its $|E_o|$ children. The other extracts up to k selections. We assume out-going edges are stored with the node, so lines 1 and 2 are $O(1)$ operations. The loop from lines 3-7 performs $|E_o|$ iterations. Each iteration, we perform two $O(1)$ operations and then insert a fixed-size entry in Q. The enqueue operation, assuming a heap implementation, requires $O(\log |Q|)$ time to insert. We insert $|E_o|$ items for a total complexity of $O(|E_o| \log |E_o|)$ time and $O(|E_o|)$ space. The loop from lines 9-18 perform $O(k)$ iterations. For each iteration, we dequeue one element of Q on Line 10. We then perform $O(1)$ operations between lines 11-13, setting some constant-size data. Finally, we sometimes add one element back into Q on Line 16. If all of o's children have k selections, then we insert a new element in Q either k or $k - 1$ times. The enqueue and dequeue operations both have an $O(\log |E_o|)$ time complexity, as the queue has at most $|E_o|$ elements in it at all times. Thus, this loop has complexity $O(k \log |E_o|)$. Combined with the first loop, the overall complexity of this algorithm is $O(|E_o| \log |E_o| + k \log |E_o|)$ time and $O(|E_o| + k)$ space.

6.3.10 Overall Find-K-Best-Selections Complexity

Given that we have now determined the complexity of all three node cases of FindKBestSelections, we are now able to compute the complexity of FindKBestSelections. This entire algorithm consists of looping over all the nodes once, and calling the appropriate sub-routine based on the type of node. Thus, the complexity of FindKBestSelections is just the number of each type of node times the complexity of that type of node. Specifically, the time required is

$$O(|L| + |A||E_a|k \log k + |O||E_o| \log |E_o| + |O|k \log |E_o|),$$

where $|E_a|$ and $|E_o|$ are the average number of children of **And** and **Or** node, respectively. If one assumes that there are approximately an equal number of each type of node and about the same number of children on average, this simplifies to

$$O(|V||E_v|k \log k + |V||E_v| \log |E_v| + |V|k \log |E_v|).$$

We substitute $|E|$ for $|V||E_v|$ as the latter just represents the number of edges in the graph, giving us

$$O(|E|k \log k + |E| \log |E_v| + |V|k \log |E_v|).$$

The space required by this algorithm is dominated by two structures, C_a and ξ. The former requires $O(k \log k)$ space. The latter requires $O(k|E|)$ space. Consequently, the FindKBestSelections algorithm requires $O(k \log k + k|E|)$ space. Note that η requires $O(k|V|)$ space, but this is never more than $O(k|E|)$ space. Likewise, β_ξ requires $O(k|E_v|)$ space, but $|E_v| < |E|$, so we can ignore this term.

6.4 Get-K-Solutions-From-Selections Algorithm

Now that we have generated our k best selections in the modified graph all the way up to the root node, we now need to follow these selections to the leaves to extract the actual solutions labels. The GetKSolutionsFromSelections algorithm assumes that the sd-DNNF graph has been augmented by

Algorithm 6.11: GetKSolutionsFromSelections(V_O, E, $\mathcal{L}_L$, η, ξ, $\#_r$)

```
1  m(*, *) ←⊥ ;
2  for i = 1 to #_r do
3      m(r, i) = i ;
4      S_k[i] = ∅ ;
5  end
6  for i = 1 to |V| do
7      v ← V_O [i] ;
8      switch v in
9          case v ∈ L
10             S_k ← GKSFSLeaf(v, m, #_r, S_k, L_L) ;
11         end
12         case v ∈ A
13             m ← GKSFSAnd(v, m, #_r, ξ) ;
14         end
15         case v ∈ O
16             m ← GKSFSOr(v, m, #_r, η) ;
17         end
18     end
19 end
20 return S_k ;
```

selections encoded by ξ and η, built upon the sd-DNNF nodes. This modified graph has up to k root nodes, each of which is a tree that defines a selection. In our modified graph, a node is a pair: $\langle n, i \rangle$. The node n is an sd-DNNF node, and i corresponds to the ith best selection for n. Thus, if $k = 3$ and the root has three selections, the best three selections start at $\langle r, 1 \rangle$, $\langle r, 2 \rangle$, and $\langle r, 3 \rangle$.

We use the marking system described at the start of the chapter, namely each node has a list of k markings $m\,(v, i)$, $i \in \{1, \ldots, k\}$. The ith marking records which of v's selections, belong to the selection that starts at $\langle r, i \rangle$. If there is not such a local selection, the marking is set to $\perp$.

Otherwise, the marking rules are the same as Appendix A in the modified graph. Namely, the leaves of the tree rooted by the ith root node include their symbols in the ith solution. The modified **And** nodes mark all of their modified children. The modified **Or** nodes mark their only modified child. The implementation of this algorithm is shown in Alg. 6.11.

Alg. 6.11 starts out on Line 1 by clearing all the marks, setting them to $\perp$. There are $O(|V|k)$ markings, of which we must clear $O(|V|\#_r)$ markings. Lines 2-5 sets the root markings and clears the solutions. A root selection $\langle r, i \rangle$ is marked as part of the ith solution. Lines 6-19 loop over all the sd-DNNF nodes, from the root to the leaves. For each node, the algorithm calls the appropriate GKSFS function, based on the type of the node. These functions move the markings down the tree corresponding to the node's selection and they set the solutions. The algorithm returns the set of solutions $\mathcal{S}_k$ on Line 20, once every modified leaf has had the opportunity to add itself to the appropriate solutions.

For example, Alg. 6.11, is responsible for extracting the 3 solutions corresponding to the 3 selections we found in the proceeding sections by running Alg. 6.2, FindKBestSelections. We have boxed two of these three selections in Figure 6-12 as well as reporting the value of each selection. The three best solutions are, in order, {"a2", "b2"}, {"a1", "b1"}, and {"a2", "b1"}.

Alg. 6.11 starts out on Line 1 by clearing all of our markings. Lines 2-5 loop once for each of the root node o1's modified nodes: $\langle \text{o1}, 1 \rangle$, $\langle \text{o1}, 2 \rangle$, and $\langle \text{o1}, 3 \rangle$. For each modified node $\langle \text{o1}, i \rangle$, the algorithm marks that modified node as part of the ith selection, and thus $m\,(\text{o1}, 1) = 1$, $m\,(\text{o1}, 2) = 2$, and $m\,(\text{o1}, 3) = 3$. The algorithm also clears $\mathcal{S}_k[i]$ for each i.

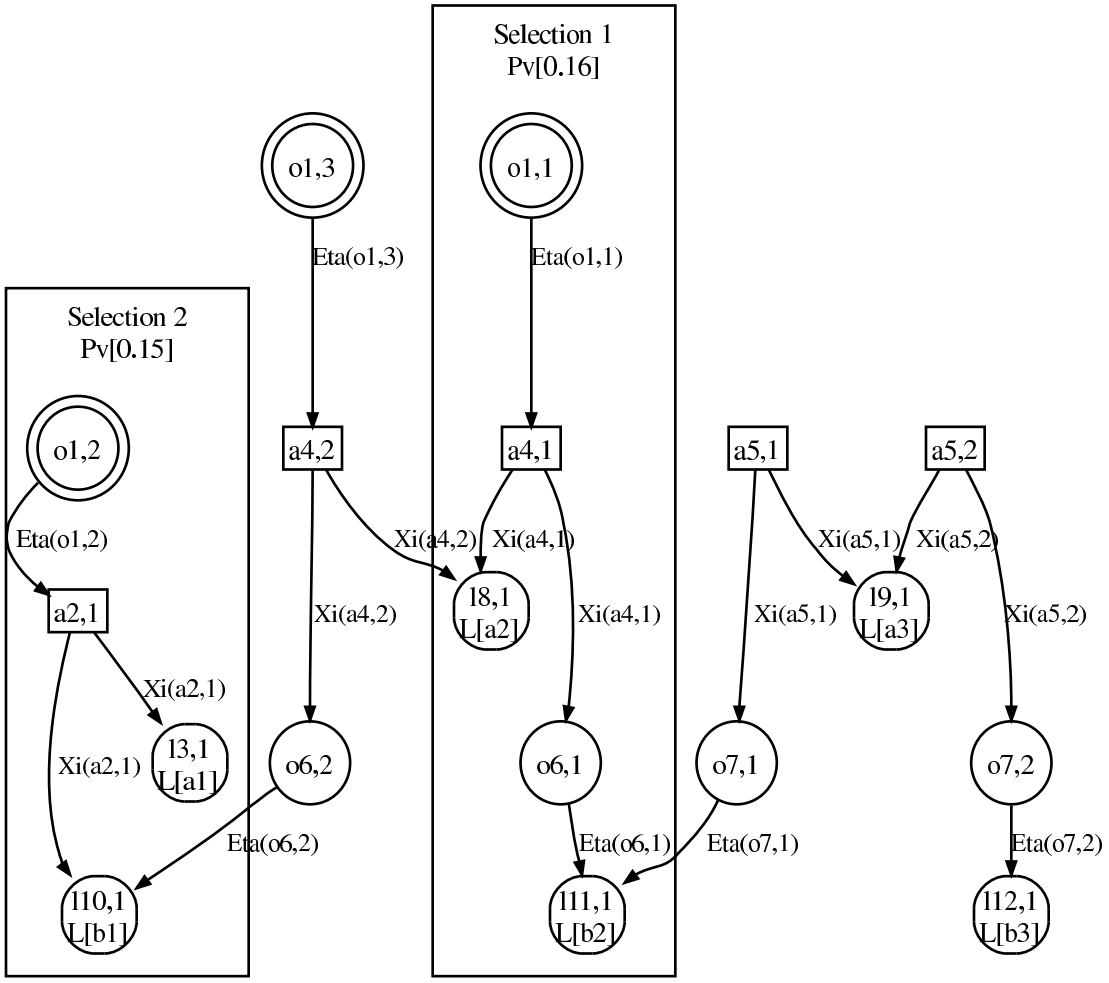

Figure 6-12: This figure shows the result of applying the GKSFS algorithm to the A-B example. We have highlighted the best 2 selections out of the 3 selections generated.

Alg. 6.11 then continues by iterating over all the sd-DNNF nodes from the root to the leaves between lines 6 and 19. The algorithm starts with the root node o1, and calls Alg. 6.14, GKSFSOr on o1. Alg. 6.14 sets $m\,(\text{a4}, 1) = 1$, $m\,(\text{a2}, 2) = 1$, and $m\,(\text{a4}, 3) = 2$. These three values mean that $\langle \text{a4}, 1\rangle$ is part of the first selection, $\langle \text{a2}, 1\rangle$ is part of the second selection, and $\langle \text{a4}, 2\rangle$ is part of the third selection, respectively.

This continues down to the leaves. One interesting aspect of this example is that two nodes are part of more than one solution, namely l8 and l10. For the node l8, for example, both $m\,(\text{l8}, 1) = 1$ and $m\,(\text{l8}, 3) = 1$, which means that l8 is part of the first and third solution. The label "a2" of l8 is thus added to $\mathcal{S}_k[1]$ and $\mathcal{S}_k[3]$.

6.4.1 Get-K-Solutions-From-Selections Leaf-case Algorithm

Algorithm 6.12: GKSFSLeaf(l, m, $\#_r$, $\mathcal{S}_k$, $\mathcal{L}_L$)

```
1 for i = 1 to #_r do
2     if m(l, i) ≠ ⊥ then
3         S_k[i] ← S_k[i] ∪ {L_L(v)} ;
4     end
5 end
6 return S_k ;
```

The leaf case, like all cases, must process all the possible root markings to see if any of them include this leaf. Since the leaf only has one modified node, $\langle l, 1\rangle$, $m\,(l, i)$ is always 1 or $\perp$. When $m\,(l, i) = 1$, the root node includes this leaf, so we add this leaf's symbol to the appropriate solution.

Time and Space Complexity The algorithm always loops $\#_r$ times, where $\#_r$ is the actual number of selections found, between 1 and k. In general, this is k. We again assume that appending symbols is an $O(1)$ operation, so for each loop, this algorithm performs an $O(1)$ operation. Thus, the time complexity of this algorithm is $O(\#_r)$. The algorithm requires only $O(1)$ local space.

Algorithm 6.13: GKSFSAnd(a, m, $\#_r$, ξ)

```
1  for i = 1 to #_r do
2      if m(a, i) ≠ ⊥ then
3          j_a ← m(a, i) ;
4          foreach ⟨a, n⟩ ∈ E do
5              j_n ← ξ(a, j_a, n) ;
6              m(n, i) ← j_n ;
7          end
8      end
9  end
10 return S_k ;
```

6.4.2 Get-K-Solutions-From-Selections And-case Algorithm

The **And** node case involves pushing each root marking that includes one of this **And** node's selections to the corresponding combination of child selections. If $m(a, i)$ is not $\perp$ on Line 2, then it specifies which modified node of a is marked for the ith solution, specifically $\langle a, j_a \rangle$. Given that $\langle a, j_a \rangle$ is part of the ith solution, we mark each modified member of the j_a combination of a, specified by ξ. For a child n of a, j_n on Line 5 is the index of the modified child node $\langle n, j_n \rangle$. ξ captures the relation that $\langle a, j_a \rangle$ is connected to $\langle n, j_n \rangle$ in our modified graph. We thus mark $\langle n, j_n \rangle$ as also being part of the solution i by setting $m(n, i) = j_n$.

Time and Space Complexity All of the operations of this algorithm are $O(1)$ within the double loop. Thus, the time complexity of the double-loop is $O(|E_a|\#_r)$, where $|E_a|$ is the number of out-going edges of a and the number of iterations of the inner loop. The algorithm requires only $O(1)$ space locally.

6.4.3 Get-K-Solutions-From-Selections Or-case Algorithm

The **Or** node case propagates each root marking that includes one of this **Or** node's modified nodes to the appropriate modified child node. If $m(o, i)$ is not $\perp$ on Line 2, then it specifies which modified node of o is marked for the solution i, specifically $\langle o, j_o \rangle$. The modified node $\langle o, j_o \rangle$ is connected to exactly one modified child node, namely $\langle n, j_n \rangle = \eta(o, j_o)$. So Line 5 marks this

Algorithm 6.14: GKSFSOr(o, m, $\#_r$, η)

```
for i = 1 to #_r do
    if m(o, i) ≠ ⊥ then
        j_o ← m(o, i) ;
        ⟨n, j_o⟩ ← η(o, j_o) ;
        m(n, i) ← j_n ;
    end
end
return S_k ;
```

modified child by setting $m(n, i) = j_n$.

Time and Space Complexity This algorithm has only one loop and everything else is looked up by index, an $O(1)$ operation, so the overall complexity is $O(\#_r)$ time. The algorithm only requires $O(1)$ local space.

6.4.4 Overall Get-K-Solutions-From-Selections Complexity

We now analyze the complexity of the whole GetKSolutionsFromSelections algorithm. The algorithm starts by clearing $O(|V|\#_r)$ markings on Line 1, requiring $O(|V|\#_r)$ time and space. Lines 2-5 set and additional $O(\#_r)$ terms, each of which is of size $O(1)$. Finally, lines 6-19 loop over all the nodes once from the root to the leaves. If $|S|$ is the size of an average solution, this loop generates $\#_r$ solutions of size $|S|$. The time complexity of the loop is

$$O(|L|\#_r + |A||E_a|\#_r + |O|\#_r).$$

The term $|A||E_a|$ represents the total number of out-going edges that have **And** node parents. If this is $O(|E|)$, then we can simplify our time complexity to $O(|E|\#_r)$. The space complexity is dominated by the space required to store the $O(|V|\#_r)$ markings.

As was the case with the FindBestSolutionFromSelection algorithm, this can be re-framed as a depth-first search, where our first step is to iterate over the $\#_r$ root nodes and then keep track of which **And** node child selection the algorithm is visiting along the path. This change reduces the

complexity of this part to $O(|\text{Sel}|\#_r)$, where $|\text{Sel}|$ is the number of nodes in a selection. The amount of space can be reduced substantially to $O(|a_{\text{Sel}}|)$, where $|a_{\text{Sel}}|$ is the largest number of **And** nodes along any path from the root to a leaf. This is the same space required to store the $k = 1$ case, as we can perform our depth-first search $\#_r$ times with the same stack.

6.5 Find-K-Best-Solutions Complexity

The overall complexity of the FindKBestSolutions algorithm is dominated by the first part of the algorithm. This chapter's algorithm has a time complexity of $O(|E|k \log k + |E| \log |E_v| + |V|k \log |E_v|)$ and a space complexity of $O(|E|k)$.

6.6 Summary

In this chapter, we presented an extension of the find-best-solution algorithm of Appendix A that is able to find up to k solutions. The extension, Alg. 6.1, has a time complexity of $O(|E|k \log k + |E| \log |E_v| + |V|k \log |E_v|)$ and a space complexity of $O(|E|k)$. We also demonstrated this algorithm on two examples, first on the simple switch example of Appendix A and then on the A-B example.

Chapter 7

Probabilistic Concurrent Constraint Automata Estimation

This chapter derives belief state update for Probabilistic Concurrent Constraint Automata (PCCA) [56] under the assumption of a uniform prior distribution over the solutions to the PCCA constraints. This chapter also contributes a reformulation of the belief state update equation and the observation probability term as an extended OCSP.

A PCCA model describes a physical system as a connected group of discrete, partially-observable, and concurrently-operating automata. A PCCA model uses probabilistic transitions to model uncertainty in the physical system. We refer to a single automaton as a Probabilistic Constraint Automaton (PCA).

Consider the simple mono-propellent propulsion system shown in Fig. 7-1. The propulsion system has a fuel tank that stores the propellent and slowly empties as it is used. There is a valve that can be opened and closed to release gas through the nozzle. When the valve is open, the thruster produces thrust. For this example we assume the valve is either open or closed (no partially open positions). The only sensor in the system is an inertial sensor that measures the acceleration produced by the thruster.

Fig. 7-2 depicts a PCA model for the fuel tank of our mono-propellent system. In our example, we model this component with two states: *Filled* and *Empty*. The tank slowly empties when there

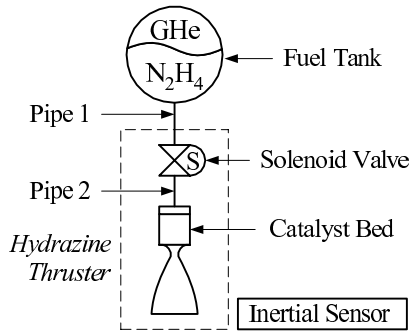

Figure 7-1: A simple mono-propellent propulsion system, with a tank, a valve, a static nozzle and an inertial sensor.

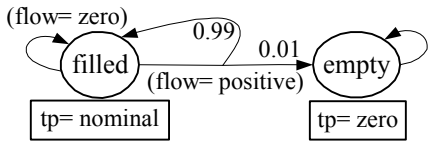

Figure 7-2: A simple PCA model of a fuel tank. The fuel tank is initially filled and eventually becomes empty as gas is taken out of the tank.

is flow out of the tank. When filled, the tank provides nominal gas pressure. When empty, the tank no longer provides gas pressure.

Fig. 7-3 depicts a PCA model for the valve of our mono-propellent system. We model this component with three states: *Open*, *Closed*, and *Stuck Closed*. The valve can be commanded to change from open to closed and from closed to open. The valve occasionally gets stuck while closed at which point it can no longer be commanded open. In our example, a stuck valve disables the propulsion system.

This chapter is divided into five sections, using the notation developed in Section 3.1. Section 7.1 reviews the standard Bayesian filter equation for Hidden Markov Models (HMM), from first principles, given an observation sequence. Section 7.2 explains the PCCA model used in this thesis to represent the physical world. Section 7.3 extends the estimation equation for HMMs to the specific case of a PCCA model. Section 7.4 then shows how an approximate version of the PCCA estimation equation can be framed as an instance of the OCSP solved in Chap. 5. Thus, we use the

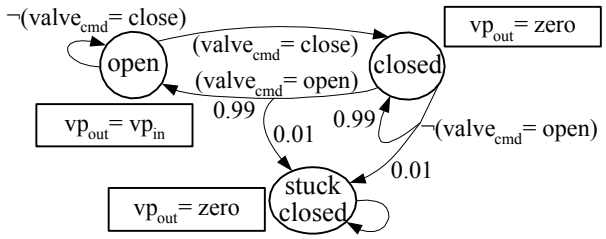

Figure 7-3: Simple PCA model of a valve. The valve can be commanded to be open or closed and has a permanent failure mode of being stuck closed.

OCSP solver of Chap. 5 to estimate the state of the PCCA model. Finally, Section 7.5 shows that the probability of a single observation given a PCCA model, used in Chap. 4 to sample observations, is also an instance of an OCSP and is solved using the OCSP solver of Chap. 5.

7.1 Review of Bayesian Filtering

This section is based on [3]. For a single state variable X and a single observation variable Y, state estimation determines a probability distribution over the current state x^{t+1} of a system, given a sequence of observations $y^{0:t+1}$, from time 0 to time $t+1$.

$$\mathbf{P}\left(x^{t+1} \middle| y^{0:t+1}\right) \tag{7.1}$$

Since we do not normally know this probability, we reformulate this probability into something we do know. Bayes' Rule states that

$$\mathbf{P}\left(A \middle| B\right) = \frac{\mathbf{P}\left(B \middle| A\right)\mathbf{P}\left(A\right)}{\mathbf{P}\left(B\right)} \tag{7.2}$$

or more generally:

$$\mathbf{P}\left(A \middle| B, C\right) = \frac{\mathbf{P}\left(B \middle| A, C\right)\mathbf{P}\left(A \middle| C\right)}{\mathbf{P}\left(B \middle| C\right)} \tag{7.3}$$

If we let $A = x^{t+1}$, $B = y^{t+1}$, and $C = y^{0:t}$, then we can rewrite $\mathbf{P}\left(x^{t+1} | \, y^{0:t+1}\right)$ as:

$$\mathbf{P}\left(x^{t+1} | \, y^{t+1}, y^{0:t}\right) = \frac{\mathbf{P}\left(y^{t+1} | \, x^{t+1}, y^{0:t}\right) \mathbf{P}\left(x^{t+1} | \, y^{0:t}\right)}{\mathbf{P}\left(y^{t+1} | \, y^{0:t}\right)} \tag{7.4}$$

If we assume that the state captures all information about the system (the Markov Property), then $\mathbf{P}\left(y^{t+1} | \, x^{t+1}, y^{0:t}\right)$ of Eq. 7.4 simplifies to $\mathbf{P}\left(y^{t+1} | \, x^{t+1}\right)$, as the observations can only depend on the current state[1].

The denominator $\mathbf{P}\left(y^{t+1} | \, y^{0:t}\right)$ of Eq. 7.4 does not depend on the state x. Thus, if we compute $\mathbf{P}\left(y^{t+1} | \, x^{t+1}\right) \mathbf{P}\left(x^{t+1} | \, y^{0:t}\right)$ for each x^{t+1}, then we can normalize these values so that they sum to one. Denoting the normalization value as α, then Eq. 7.4 becomes

$$\mathbf{P}\left(x^{t+1} | \, y^{0:t+1}\right) = \alpha \mathbf{P}\left(y^{t+1} | \, x^{t+1}\right) \mathbf{P}\left(x^{t+1} | \, y^{0:t}\right) \tag{7.5}$$

We cannot directly compute $\mathbf{P}\left(x^{t+1} | \, y^{0:t}\right)$, instead we compute it through marginalization:

$$\mathbf{P}\left(x^{t+1} | \, y^{0:t}\right) = \sum_{x^t \in \mathbb{D}_{X^t}} \mathbf{P}\left(x^{t+1}, x^t | \, y^{0:t}\right) \tag{7.6}$$

Using the conditional probability rule

$$\mathbf{P}\left(A, B | \, C\right) = \mathbf{P}\left(A | \, B, C\right) \mathbf{P}\left(B | \, C\right) \tag{7.7}$$

We expand $\mathbf{P}\left(x^{t+1}, x^t | \, y^{0:t}\right)$ to:

$$\mathbf{P}\left(x^{t+1}, x^t | \, y^{0:t}\right) = \mathbf{P}\left(x^{t+1} | \, x^t, y^{0:t}\right) \mathbf{P}\left(x^t | \, y^{0:t}\right) \tag{7.8}$$

Using the Markov property, $\mathbf{P}\left(x^{t+1} | x^t, y^{0:t}\right)$ simplifies to $\mathbf{P}\left(x^{t+1} | x^t\right)$ in Eq. 7.8. With this

[1] If this assumption does not hold, then there is some process operating in our system that has a state that we failed to model with x. If we incorporate this state into our state variable, then this assumption holds.

simplification and substituting Eq. 7.8 into Eq. 7.6 we obtain:

$$\mathbf{P}\left(x^{t+1}|\, y^{0:t}\right) = \sum_{x^t \in \mathbb{D}_{X^t}} \mathbf{P}\left(x^{t+1}|\, x^t\right) \mathbf{P}\left(x^t|\, y^{0:t}\right) \tag{7.9}$$

Substituting this Eq. 7.9 into Eq. 7.5 yields our final result, the Bayes Filter update Equation:

$$\mathbf{P}\left(x^{t+1}|\, y^{0:t+1}\right) = \alpha \mathbf{P}\left(y^{t+1}|\, x^{t+1}\right) \sum_{x^t \in \mathbb{D}_{X^t}} \mathbf{P}\left(x^{t+1}|\, x^t\right) \mathbf{P}\left(x^t|\, y^{0:t}\right), \tag{7.10}$$

Eq. 7.10 is a recursive equation: $\mathbf{P}\left(x^{t+1}|\, y^{0:t+1}\right)$ in terms of $\mathbf{P}\left(x^t|\, y^{0:t}\right)$. The term $\mathbf{P}\left(y^{t+1}|\, x^{t+1}\right)$ is the probability of observing y^{t+1} in the state x^{t+1}. The term $\mathbf{P}\left(x^{t+1}|\, x^t\right)$ is the transition probability distribution, the probability of going from state x^t to x^{t+1}.

A *Hidden Markov Model* (HMM) [3] is a model that specifies the two distributions $\mathbf{P}\left(y^{t+1}|\, x^{t+1}\right)$ and $\mathbf{P}\left(x^{t+1}|\, x^t\right)$ explicitly, where x and y are finite domain variables. Thus, when using an HMM model, Eq. 7.10 can be used directly to estimate the probability of each state.

These equations have thus far been developed for a single state variable x and a single observation variable y. A natural extension is to partition both the state and observation into multiple variables. This extension changes the meaning of the state to a vector of assignments instead of a single assignment and likewise the observation to a vector of assignments, but the equations are otherwise the same:

$$\mathbf{P}\left(\mathbf{x}^{t+1}|\, \mathbf{y}^{0:t+1}\right) = \alpha \mathbf{P}\left(\mathbf{y}^{t+1}|\, \mathbf{x}^{t+1}\right) \sum_{\mathbf{x}^t \in \mathbb{D}_{\mathbf{X}^t}} \mathbf{P}\left(\mathbf{x}^{t+1}|\, \mathbf{x}^t\right) \mathbf{P}\left(\mathbf{x}^t|\, \mathbf{y}^{0:t}\right) \tag{7.11}$$

7.2 The PCCA Model

In this section we review the *Probabilistic Concurrent Constraint Automata* (PCCA) model used by this thesis to model the plant. A PCCA model compactly encodes a discrete state model through concurrency and finite domain constraints. The model is factored into a set of *Probabilistic Constraint Automata* (PCA), each of which encapsulates a component's behavior. Components describe their behavior qualitatively with constraints on how their inputs and outputs are related, based on the

"mode" of the component. Constraints are also used to describe how inputs and outputs of components are related. This encapsulation allows for the definition of generic, re-usable components that can be connected together appropriately to model a specific system. In general the number of states $\mathbf{x}$ is too large to enumerate explicitly, and thus we are interested in approximating $\mathbf{P}\left(\mathbf{x}^{t+1} | \, \mathbf{y}^{0:t+1}\right)$. In this section we define our PCCA model. In the rest of this chapter we explain how to approximate $\mathbf{P}\left(\mathbf{x}^{t+1} | \, \mathbf{y}^{0:t+1}\right)$.

A PCCA is a composition of PCA A_a, each of which is defined by a triple $\langle \mathbf{X}_a, \mathbb{M}_a, \mathbb{T}_a \rangle$:

1. $\mathbf{X}_a = \mathbf{M}_a \cup \mathbf{C}_a \cup \mathbf{D}_a$ is a finite set of discrete variables, which completely describe the component. All $X \in \mathbf{X}_a$ have a finite domain $\mathbb{D}_X$. $\mathbf{M}$, $\mathbf{C}$, and $\mathbf{D}$ correspond to *mode*, *control*, and *dependent* variables, respectively. The mode variables are the estimated variables. The control variables are assumed to be issued by a local controller and thus their values are known and reliable. We denote a value of a command variable with the special notation μ, as is customary, instead of c. The dependent variables are the intermediate variables needed to define the behavior of a single component; dependent variables are state-less. At each point in time t, we may observe the value of some $\mathbf{O}_a \subseteq \mathbf{D}_a$. In other words, we may only receive observations from some subset of our sensors, where those that do not provide observations have either failed or do not generate observations at every time step. Some $\mathbf{D}_a$ may never be observed. We denote the complete set of full assignments to variables $\mathbf{X}$ as $\mathbb{D}_{\mathbf{X}}$, and the set of all possible constraints on variables $\mathbf{X}$ as $\mathbb{C}\left(\mathbf{X}\right)$.

2. $\mathbb{M}_a : \mathbb{D}_{\mathbf{M}_a^t} \rightarrow \mathbb{C}\left(\mathbf{D}_a^t\right)$, the *modal constraints*, map each mode variable to a constraint that must hold true when the component is within that mode.

3. $\tau_a : \mathbb{D}_{\mathbf{M}_a^t \cup \mathbf{C}_a^t \cup \mathbf{D}_a^t \cup \mathbf{M}_a^{t+1}} \rightarrow \mathbb{R}\left[0, 1\right]$ represents a guarded, probabilistic transition function. Consider an evaluation of the transition $\tau_a(\mathbf{m}_a^t, \boldsymbol{\mu}_a^t, \mathbf{d}_a^t, \mathbf{m}_a^{t+1})$. $\mathbf{m}_a^t$ represents the source mode of the component at time t. $\mathbf{m}_a^{t+1}$ is the target mode of the component at time $t+1$, after the transition. τ_a evaluates to non-zero when this transition can occur. The transition function τ_a specifies the probability $\mathbf{P}\left(\mathbf{m}_a^{t+1} | \mathbf{m}_a^t, \boldsymbol{\mu}_a^t, \mathbf{d}_a^t\right)$. We assume τ_a is time invariant.

A PCCA model $\mathcal{P}$ is defined by the triple $\mathcal{P} = \langle \mathcal{A}, \mathbf{X}, \mathbb{Q} \rangle$:

1. $\mathcal{A} = \{A_1, \ldots, A_n\}$ is the finite set of PCAs; one PCA for each component.

2. $\mathbf{X} = \bigcup_{a=1..n} \mathbf{X}_a$ is the set of all variables defined in $\mathcal{A}$.

3. $\mathbb{Q}^t \in \mathbb{C}\left(\mathbf{X}^t\right)$ is a constraint over all variables at one time point, and captures the interconnections between components. We assume $\mathbb{Q}^t$ is time invariant.

For this thesis, we reformulate our PCCA definition to the triple $\mathcal{P} = \langle \mathbf{X}, \Phi, \mathbb{T} \rangle$:

1. $\mathbf{X}$ as before

2. $$\begin{aligned}\Phi =& \mathbb{Q}^{t:t+1} \wedge \left(\bigwedge_{\mathbf{m}_a^t} \mathbf{m}_a^t \Rightarrow \mathbb{M}_a(\mathbf{m}_a^t) \right) \wedge \left(\bigwedge_{\mathbf{m}_a^{t+1}} \mathbf{m}_a^{t+1} \Rightarrow \mathbb{M}_a(\mathbf{m}_a^{t+1}) \right) \wedge \\ & \left(\bigwedge_{\mathbf{m}_a^t, \boldsymbol{\mu}_a^t, \mathbf{d}_a^t, \mathbf{m}_a^{t+1}} (\tau_a(\mathbf{m}_a^t, \boldsymbol{\mu}_a^t, \mathbf{d}_a^t, \mathbf{m}_a^{t+1}) > 0) \right)\end{aligned}$$

3. $\mathbb{T} = \{\tau_a\}$

With this reformulation, our PCCA model is a collection of variables, a hard constraint over two time points, and a set of probabilistic transitions.

7.2.1 Example PCCA Model of the Propulsion System

To illustrate our PCCA model formulation, consider again our simplified monopropellant propulsion system. The schematic of the propulsion subsystem is shown in Figure 7-1. We model this propulsion subsystem as a set of two components: a fuel tank and a solenoid valve. We assume that a properly opened solenoid valve always leads to a nominal inertial sensor measurement while the tank still has fuel.

Fuel Tank: The fuel tank PCA model A_{tank} is shown graphically in Figure 7-2. A_{tank} is defined by the triple $A_{\text{tank}} = \langle \mathbf{X}_{\text{tank}}, \mathbb{M}_{\text{tank}}, \tau_{\text{tank}} \rangle$. The variables are $\mathbf{X}_{\text{tank}} = \{\text{tank}, \text{flow}, \text{tp}\}$, where the fuel tank's state, represented by variable tank, resides in one of two discrete modes, $\mathbb{D}_{\text{tank}} =$

$\{\text{filled}, \text{empty}\}$. The variable flow describes whether fuel is flowing from the tank, with the domain $\mathbb{D}_{\text{flow}} = \{\text{zero}, \text{positive}\}$. The variable tp describes whether the tank is pressurized, which indicates whether or not the fuel tank contains fuel. It has the domain $\mathbb{D}_{\text{tp}} = \{\text{zero}, \text{nominal}\}$. The modal constraint $\mathbb{M}_{\text{tank}}$ and transition τ_{tank} are:

$\mathbb{M}_{\text{tank}}$

tank^t	C
filled	$\{(\text{tp}^t = \text{nominal})\}$
empty	$\{(\text{tp}^t = \text{zero})\}$

τ_{tank}

tank^t	flow^t	tank^{t+1}	p
filled	zero	filled	1
filled	positive	filled	0.99
filled	positive	empty	0.01
empty	——	empty	1

Solenoid Valve: The solenoid valve PCA model A_{valve} is shown graphically in Fig. 7-3, and is defined in a manner similar to the Fuel Tank. The variables are $\mathbf{X}_{\text{valve}} = \{\text{valve}, \text{vp}_{\text{in}}, \text{vp}_{\text{out}}, \text{valve}_{\text{cmd}}\}$, where the solenoid valve's state, represented by the variable valve, resides in one of three discrete modes, $\mathbb{D}_{\text{valve}} = \{\text{open}, \text{closed}, \text{stuck (closed)}\}$. The variable vp_{in} describes the pressure at the valve inlet; the variable vp_{out} describes the pressure at the valve outlet, reflecting whether fuel is flowing. Both variables have domain $\mathbb{D}_{\text{vp}_{\text{in}}} = \mathbb{D}_{\text{vp}_{\text{out}}} = \{\text{zero}, \text{nominal}\}$. The variable $\text{valve}_{\text{cmd}}$ describes the commands that may be issued to the valve: open and close. A command may also not be issued, so $\mathbb{D}_{\text{valve}_{\text{cmd}}} = \{\text{open}, \text{close}, \text{no-cmd}\}$. As can be seen in Fig. 7-3, the modal constraint $\mathbb{M}_{\text{valve}}$ and the transition τ_{valve} are:

$\mathbb{M}_{\text{valve}}$

valve^t	C
open	$\{\text{vp}^t_{\text{out}} = \text{vp}^t_{\text{in}}\}$
closed	$\{(\text{vp}^t_{\text{out}} = \text{zero})\}$
stuck	$\{(\text{vp}^t_{\text{out}} = \text{zero})\}$

τ_{valve}

valve^t	$\text{valve}^t_{\text{cmd}}$	valve^{t+1}	p
closed	open	open	0.99
closed	$\neg$ open	closed	0.99
closed	——	stuck	0.01
open	$\neg$ close	open	1
open	close	closed	1
stuck	——	stuck	1

7.2.2 Combined PCCA Model

Combining these components, the PCCA model $\mathcal{P}$ is defined by the three components:

1. $\mathcal{A} = \{A_{\text{tank}}, A_{\text{valve}}\}$

2. $\mathbf{X} = \mathbf{X}_{\text{tank}} \cup \mathbf{X}_{\text{valve}} = \{\text{tank}, \text{flow}, \text{tp}, \text{valve}, \text{vp}_{\text{in}}, \text{vp}_{\text{out}}, \text{valve}_{\text{cmd}}\}$

3. $\mathbb{Q}$ connects tp to vp_{in} and vp_{out} to flow. The components are connected through a single pressure variable. There is flow when the pressure at the output of the valve is not zero:

$$\mathbb{Q}^t = \left\{ \text{tp}^t = \text{vp}^t_{\text{in}}, \begin{array}{c|c} \text{vp}^t_{\text{out}} & \text{flow}^t \\ \hline \text{zero} & \text{zero} \\ \text{nominal} & \text{positive} \end{array} \right\}.$$

The reformulated version of this model is:

1. $\mathbf{X} = \{\text{tank}, \text{flow}, \text{tp}, \text{valve}, \text{vp}_{\text{in}}, \text{vp}_{\text{out}}, \text{valve}_{\text{cmd}}\}$

2. Let $\Psi^t = \left\{ \mathbb{Q}^t, \begin{array}{c|c} \text{tank}^t & \text{tp}^t \\ \hline \text{filled} & \text{nominal} \\ \text{empty} & \text{zero} \end{array}, \begin{array}{c|c|c} \text{valve}^t & \text{vp}^t_{\text{in}} & \text{vp}^t_{\text{out}} \\ \hline \text{open} & \text{zero} & \text{zero} \\ \text{open} & \text{nominal} & \text{nominal} \\ \text{closed} & \text{——} & \text{zero} \\ \text{stuck} & \text{——} & \text{zero} \end{array} \right\}$

Then $\Phi = \left\{ \Psi^t, \Psi^{t+1}, \begin{array}{c|c|c} \text{tank}^t & \text{flow}^t & \text{tank}^{t+1} \\ \hline \text{filled} & \text{——} & \text{filled} \\ \text{filled} & \text{positive} & \text{empty} \\ \text{empty} & \text{——} & \text{empty} \end{array}, \begin{array}{c|c|c} \text{valve}^t & \text{valve}^t_{\text{cmd}} & \text{valve}^{t+1} \\ \hline \text{closed} & \text{open} & \text{open} \\ \text{closed} & \neg\,\text{open} & \text{closed} \\ \text{closed} & \text{——} & \text{stuck} \\ \text{open} & \neg\,\text{close} & \text{open} \\ \text{open} & \text{close} & \text{closed} \\ \text{stuck} & \text{——} & \text{stuck} \end{array} \right\}$

3. $\mathbb{T} = \left\{ \begin{array}{c|c|c|c} \text{tank}^t & \text{flow}^t & \text{tank}^{t+1} & p \\ \hline \text{filled} & \text{zero} & \text{filled} & 1 \\ \text{filled} & \text{positive} & \text{filled} & 0.99 \\ \text{filled} & \text{positive} & \text{empty} & 0.01 \\ \text{empty} & \text{——} & \text{empty} & 1 \end{array}, \begin{array}{c|c|c|c} \text{valve}^t & \text{valve}^t_{\text{cmd}} & \text{valve}^{t+1} & p \\ \hline \text{closed} & \text{open} & \text{open} & 0.99 \\ \text{closed} & \neg\,\text{open} & \text{closed} & 0.99 \\ \text{closed} & \text{——} & \text{stuck} & 0.01 \\ \text{open} & \neg\,\text{close} & \text{open} & 1 \\ \text{open} & \text{close} & \text{closed} & 1 \\ \text{stuck} & \text{——} & \text{stuck} & 1 \end{array} \right\}$

7.3 PCCA Belief State Estimation

In this section we specialize the Bayesian Filter equation, Eq. 7.11, to a PCCA Filter. Thus $\mathbf{x} = \mathbf{m}$, $\mathbf{y} = \mathbf{o}$, and we incorporate the commands $\boldsymbol{\mu}$. We estimate the PCCA model's mode given the observations and commands, hence the equivalent formulation of Eq. 7.1 is:

$$\mathbf{P}\left(\mathbf{m}^{t+1} | \mathbf{o}^{0:t+1}, \boldsymbol{\mu}^{0:t}\right) \tag{7.12}$$

Recall that we denote an observation by $\mathbf{o}^t$ such that $\text{vars}(\mathbf{o}^t) \subseteq \mathbf{D}^t$.[2] Let us denote the remaining dependent variables by $\mathbf{U}^t = \mathbf{D}^t \setminus \text{vars}(\mathbf{o}^t)$ as the unspecified variables. We denote an assignment to the unspecified variables as $\mathbf{u}^t$.

We now derive the recursive belief state update equations for PCCA models, specifically we determine $\mathbf{P}\left(\mathbf{m}^{t+1}|\mathbf{o}^{0:t+1}, \boldsymbol{\mu}^{0:t}\right)$ as a function of $\mathbf{P}\left(\mathbf{m}^{t}|\mathbf{o}^{0:t}, \boldsymbol{\mu}^{0:t-1}\right)$. The final result is

$$B^{t+1}\left(\mathbf{m}^{t+1}\right) = \alpha \sum_{\mathbf{m}^t \in \mathbb{D}_{\mathbf{M}^t}} \frac{\sum_{\mathbf{u}^{t:t+1} \in \mathbb{D}_{\mathbf{u}^{t:t+1}}} \mathcal{C}\left(\mathbf{x}^{t:t+1}\right) \tau\left(\mathbf{x}^{-}\right)}{\sum_{\mathbf{o}_i^{t+1} \in \mathbb{D}_{\mathbf{O}^{t+1}}} \sum_{\mathbf{u}^{t:t+1} \in \mathbb{D}_{\mathbf{U}^{t:t+1}}} \mathcal{C}\left(\mathbf{x}_i^{t:t+1}\right)} B^t\left(\mathbf{m}^t\right) \tag{7.13}$$

Starting with the Bayesian Filter Equation:

$$\begin{aligned}
&\mathbf{P}\left(\mathbf{m}^{t+1}|\mathbf{o}^{0:t+1}, \boldsymbol{\mu}^{0:t}\right) \\
&= \sum_{\mathbf{m}^t \in \mathbb{D}_{\mathbf{M}^t}} \sum_{\mathbf{u}^t \in \mathbb{D}_{\mathbf{U}^t}} \mathbf{P}\left(\mathbf{m}^{t:t+1}, \mathbf{u}^t|\mathbf{o}^{0:t+1}, \boldsymbol{\mu}^{0:t}\right) && (7.14)\\
&= \alpha \sum_{\mathbf{m}^t \in \mathbb{D}_{\mathbf{M}^t}} \sum_{\mathbf{u}^t \in \mathbb{D}_{\mathbf{U}^t}} \mathbf{P}\left(\mathbf{m}^{t+1}, \mathbf{u}^t, \mathbf{o}^{t+1}|\mathbf{m}^t, \mathbf{o}^{0:t}, \boldsymbol{\mu}^{0:t}\right) \mathbf{P}\left(\mathbf{m}^t|\mathbf{o}^{0:t}, \boldsymbol{\mu}^{0:t-1}\right) && (7.15)\\
&= \alpha \sum_{\mathbf{m}^t \in \mathbb{D}_{\mathbf{M}^t}} \sum_{\mathbf{u}^t \in \mathbb{D}_{\mathbf{U}^t}} \mathbf{P}\left(\mathbf{m}^{t+1}, \mathbf{u}^t, \mathbf{o}^{t+1}|\mathbf{m}^t, \mathbf{o}^{t}, \boldsymbol{\mu}^{t}\right) \mathbf{P}\left(\mathbf{m}^t|\mathbf{o}^{0:t}, \boldsymbol{\mu}^{0:t-1}\right) && (7.16)
\end{aligned}$$

Where $\frac{1}{\alpha} = \mathbf{P}\left(\mathbf{o}^{t+1}|\mathbf{o}^{0:t}, \boldsymbol{\mu}^{0:t}\right)$. Let us define

$$B^t\left(\mathbf{m}^t\right) = \mathbf{P}\left(\mathbf{m}^t|\mathbf{o}^{0:t}, \boldsymbol{\mu}^{0:t-1}\right) \tag{7.17}$$

Then

$$B^{t+1}\left(\mathbf{m}^{t+1}\right) = \alpha \sum_{\mathbf{m}^t \in \mathbb{D}_{\mathbf{M}^t}} \sum_{\mathbf{u}^t \in \mathbb{D}_{\mathbf{U}^t}} \mathbf{P}\left(\mathbf{m}^{t+1}, \mathbf{u}^t, \mathbf{o}^{t+1}|\mathbf{m}^t, \mathbf{o}^{t}, \boldsymbol{\mu}^{t}\right) B^t\left(\mathbf{m}^t\right) \tag{7.18}$$

[2]Note that we assume that all command variables are always observed, since we control their value, and that state variables are never observed. We can relax this latter constraint by allowing $\mathbf{O}^t$ to be a subset of both $\mathbf{M}^t$ and $\mathbf{D}^t$. This change makes the problem of estimating modes easier, as we are told the value of some of them. Otherwise, this does not change the derivation significantly, hence we ignore this option here.

As was the case in Section 7.1, our transitions are Markov. Let

$$\tau\left(\mathbf{m}^{t:t+1}, \mathbf{o}^t, \boldsymbol{\mu}^t, \mathbf{u}^t\right) = \prod_{m_a^{t:t+1} \in \mathbf{m}^{t:t+1}} \tau_a\left(\left(\mathbf{m}^{t:t+1}, \mathbf{o}^t, \boldsymbol{\mu}^t, \mathbf{u}^t\right) \Downarrow_{\mathbf{X}_a^{t:t+1}}\right) \tag{7.19}$$

$$= \mathbf{P}\left(\mathbf{m}^{t+1} | \mathbf{m}^t, \mathbf{u}^t, \mathbf{o}^t, \boldsymbol{\mu}^t\right) \tag{7.20}$$

and

$$\mathcal{C}\left(\mathbf{m}^{t:t+1}, \mathbf{o}^{t:t+1}, \boldsymbol{\mu}^t, \mathbf{u}^{t:t+1}\right) = \begin{cases} 1 & \text{Consistent}\left(\Phi^{t:t+1} \wedge \mathbf{m}^{t:t+1} \wedge \mathbf{o}^{t:t+1} \wedge \boldsymbol{\mu}^t \wedge \mathbf{u}^{t:t+1}\right) \\ 0 & \text{Otherwise} \end{cases} \tag{7.21}$$

Note that Φ factors into two sets of constraints, given $\mathbf{m}^{t+1}$: those that depend on the variables $\left(\mathbf{M}^{t:t+1}, \mathbf{O}^t, \mathbf{C}^t, \mathbf{U}^t\right)$ and those that depend on $\left(\mathbf{M}^{t+1}, \mathbf{O}^{t+1}, \mathbf{U}^{t+1}\right)$. Thus, given $\mathbf{m}^{t+1}$, we can partition $\mathcal{C}$ into two equations: $\mathcal{C}^-\left(\mathbf{m}^{t:t+1}, \mathbf{o}^t, \mathbf{u}^t, \boldsymbol{\mu}^t\right)$ and $\mathcal{C}^+\left(\mathbf{m}^{t+1}, \mathbf{o}^{t+1}, \mathbf{u}^{t+1}\right)$.

Returning to Eq. 7.18 and substituting in Eqs. 7.10 and 7.11, we get

$$B^{t+1}\left(\mathbf{m}^{t+1}\right) = \alpha \mathbf{P}\left(\mathbf{o}^{t+1} | \mathbf{m}^{t+1}\right) \sum_{\mathbf{m}^t \in \mathbb{D}_{\mathbf{M}^t}} \sum_{\mathbf{u}^t \in \mathbb{D}_{\mathbf{U}^t}} \mathbf{P}\left(\mathbf{m}^{t+1}, \mathbf{u}^t | \mathbf{m}^t, \mathbf{o}^t, \boldsymbol{\mu}^t\right) B^t\left(\mathbf{m}^t\right) \tag{7.22}$$

We define the observation probability in terms of counting of the number of ways one can arrive at the observation over the number of ways to get any observation, in terms of solutions to the theory Φ. We use this approach as we have no information as to the likelihood of each solution:

$$\mathbf{P}\left(\mathbf{o}^{t+1} | \mathbf{m}^{t+1}\right) = \frac{\sum_{\mathbf{u}^{t+1} \in \mathbb{D}_{\mathbf{U}^{t+1}}} \mathcal{C}^+\left(\mathbf{m}^{t+1}, \mathbf{o}^{t+1}, \mathbf{u}^{t+1}\right)}{\sum_{\mathbf{o}_i^{t+1} \in \mathbb{D}_{\mathbf{O}^{t+1}}} \sum_{\mathbf{u}^{t+1} \in \mathbb{D}_{\mathbf{U}^{t+1}}} \mathcal{C}^+\left(\mathbf{m}^{t+1}, \mathbf{o}_i^{t+1}, \mathbf{u}^{t+1}\right)} \tag{7.23}$$

We break up the term $\mathbf{P}\left(\mathbf{m}^{t+1}, \mathbf{u}^t | \mathbf{m}^t, \mathbf{o}^t, \boldsymbol{\mu}^t\right)$ into two parts using the conditional probability rule $\mathbf{P}\left(A, B | C\right) = \mathbf{P}\left(A | B, C\right) \mathbf{P}\left(B | C\right)$:

$$\mathbf{P}\left(\mathbf{m}^{t+1}, \mathbf{u}^t | \mathbf{m}^t, \mathbf{o}^t, \boldsymbol{\mu}^t\right) = \mathbf{P}\left(\mathbf{m}^{t+1} | \mathbf{m}^t, \mathbf{u}^t, \mathbf{o}^t, \boldsymbol{\mu}^t\right) \mathbf{P}\left(\mathbf{u}^t | \mathbf{m}^t, \mathbf{o}^t, \boldsymbol{\mu}^t\right) \tag{7.24}$$

The first term on the right is the probability of transitioning, given all past information as given by the PCCA model, Eq. 7.19. The second is unspecified by the model, and thus we again use a uniform probability distribution over the solutions to the constraints:

$$\mathbf{P}\left(\mathbf{u}^t|\mathbf{m}^t,\mathbf{o}^t,\boldsymbol{\mu}^t\right)=\frac{\mathcal{C}^-\left(\mathbf{m}^{t:t+1},\mathbf{o}^t,\mathbf{u}^t,\boldsymbol{\mu}^t\right)}{\sum\limits_{\mathbf{u}_i^t\in\mathbb{D}_{\mathbf{U}^t}}\mathcal{C}^-\left(\mathbf{m}^{t:t+1},\mathbf{o}^t,\mathbf{u}_i^t,\boldsymbol{\mu}^t\right)} \tag{7.25}$$

Substituting Eqs. 7.25 and 7.19 into Eq. 7.24 yields:

$$\mathbf{P}\left(\mathbf{m}^{t+1},\mathbf{u}^t|\mathbf{m}^t,\mathbf{o}^t,\boldsymbol{\mu}^t\right)=\frac{\tau\left(\mathbf{m}^{t:t+1},\mathbf{o}^t,\mathbf{u}^t,\boldsymbol{\mu}^t\right)\mathcal{C}^-\left(\mathbf{m}^{t:t+1},\mathbf{o}^t,\mathbf{u}^t,\boldsymbol{\mu}^t\right)}{\sum\limits_{\mathbf{u}_i^t\in\mathbb{D}_{\mathbf{U}^t}}\mathcal{C}^-\left(\mathbf{m}^{t:t+1},\mathbf{o}^t,\mathbf{u}_i^t,\boldsymbol{\mu}^t\right)} \tag{7.26}$$

If we let

$$\tau^-\left(\mathbf{m}^{t:t+1},\mathbf{o}^t,\mathbf{u}^t,\boldsymbol{\mu}^t\right)=\tau\left(\mathbf{m}^{t:t+1},\mathbf{o}^t,\mathbf{u}^t,\boldsymbol{\mu}^t\right)\mathcal{C}^-\left(\mathbf{m}^{t:t+1},\mathbf{o}^t,\mathbf{u}^t,\boldsymbol{\mu}^t\right) \tag{7.27}$$

and substitute Eqs. 7.26 and 7.23 into Eq. 7.22, we get:

$$B^{t+1}\left(\mathbf{m}^{t+1}\right)=\alpha\left(\frac{\sum\limits_{\mathbf{u}^{t+1}\in\mathbb{D}_{\mathbf{U}^{t+1}}}\mathcal{C}^+\left(\mathbf{m}^{t+1},\mathbf{o}^{t+1},\mathbf{u}^{t+1}\right)}{\sum\limits_{\mathbf{o}_i^{t+1}\in\mathbb{D}_{\mathbf{O}^{t+1}}}\sum\limits_{\mathbf{u}^{t+1}\in\mathbb{D}_{\mathbf{U}^{t+1}}}\mathcal{C}^+\left(\mathbf{m}^{t+1},\mathbf{o}_i^{t+1},\mathbf{u}^{t+1}\right)}\right)\cdot\left(\sum\limits_{\mathbf{m}^t\in\mathbb{D}_{\mathbf{M}^t}}\sum\limits_{\mathbf{u}^t\in\mathbb{D}_{\mathbf{U}^t}}\frac{\tau^-\left(\mathbf{m}^{t:t+1},\mathbf{o}^t,\mathbf{u}^t,\boldsymbol{\mu}^t\right)}{\sum\limits_{\mathbf{u}_i^t\in\mathbb{D}_{\mathbf{U}^t}}\mathcal{C}^-\left(\mathbf{m}^{t:t+1},\mathbf{o}^t,\mathbf{u}_i^t,\boldsymbol{\mu}^t\right)}B^t\left(\mathbf{m}^t\right)\right) \tag{7.28}$$

$$=\alpha\left(\frac{\sum\limits_{\mathbf{u}^{t+1}\in\mathbb{D}_{\mathbf{U}^{t+1}}}\mathcal{C}^+\left(\mathbf{m}^{t+1},\mathbf{o}^{t+1},\mathbf{u}^{t+1}\right)}{\sum\limits_{\mathbf{o}_i^{t+1}\in\mathbb{D}_{\mathbf{O}^{t+1}}}\sum\limits_{\mathbf{u}^{t+1}\in\mathbb{D}_{\mathbf{U}^{t+1}}}\mathcal{C}^+\left(\mathbf{m}^{t+1},\mathbf{o}_i^{t+1},\mathbf{u}^{t+1}\right)}\right)\cdot\left(\sum\limits_{\mathbf{m}^t\in\mathbb{D}_{\mathbf{M}^t}}\frac{\sum\limits_{\mathbf{u}^t\in\mathbb{D}_{\mathbf{U}^t}}\tau^-\left(\mathbf{m}^{t:t+1},\mathbf{o}^t,\mathbf{u}^t,\boldsymbol{\mu}^t\right)}{\sum\limits_{\mathbf{u}^t\in\mathbb{D}_{\mathbf{U}^t}}\mathcal{C}^-\left(\mathbf{m}^{t:t+1},\mathbf{o}^t,\mathbf{u}^t,\boldsymbol{\mu}^t\right)}B^t\left(\mathbf{m}^t\right)\right) \tag{7.29}$$

If we let

$$\mathbf{x}^{+} = \mathbf{m}^{t+1} \cup \mathbf{o}^{t+1} \cup \mathbf{u}^{t+1}$$
$$\mathbf{x}_i^{+} = \mathbf{m}^{t+1} \cup \mathbf{o}_i^{t+1} \cup \mathbf{u}^{t+1}$$
$$\mathbf{x}^{-} = \mathbf{m}^{t:t+1} \cup \mathbf{o}^{t} \cup \boldsymbol{\mu}^{t} \cup \mathbf{u}^{t}$$

We can then simplify Eq. 7.29 and combine terms:

$$= \alpha \frac{\sum\limits_{\mathbf{u}^{t+1} \in \mathbb{D}_{\mathbf{U}^{t+1}}} \mathcal{C}^{+}\left(\mathbf{x}^{+}\right)}{\sum\limits_{\mathbf{o}_i^{t+1} \in \mathbb{D}_{\mathbf{O}^{t+1}}} \sum\limits_{\mathbf{u}^{t+1} \in \mathbb{D}_{\mathbf{U}^{t+1}}} \mathcal{C}^{+}\left(\mathbf{x}_i^{+}\right)} \sum\limits_{\mathbf{m}^{t} \in \mathbb{D}_{\mathbf{M}^{t}}} \frac{\sum\limits_{\mathbf{u}^{t} \in \mathbb{D}_{\mathbf{U}^{t}}} \tau^{-}\left(\mathbf{x}^{-}\right)}{\sum\limits_{\mathbf{u}^{t} \in \mathbb{D}_{\mathbf{U}^{t}}} \mathcal{C}^{-}\left(\mathbf{x}^{-}\right)} B^{t}\left(\mathbf{m}^{t}\right) \tag{7.30}$$

$$= \alpha \sum_{\mathbf{m}^{t} \in \mathbb{D}_{\mathbf{M}^{t}}} \frac{\left(\sum\limits_{\mathbf{u}^{t+1} \in \mathbb{D}_{\mathbf{U}^{t+1}}} \mathcal{C}^{+}\left(\mathbf{x}^{+}\right)\right)\left(\sum\limits_{\mathbf{u}^{t} \in \mathbb{D}_{\mathbf{U}^{t}}} \tau^{-}\left(\mathbf{x}^{-}\right)\right)}{\left(\sum\limits_{\mathbf{o}_i^{t+1} \in \mathbb{D}_{\mathbf{O}^{t+1}}} \sum\limits_{\mathbf{u}^{t+1} \in \mathbb{D}_{\mathbf{U}^{t+1}}} \mathcal{C}^{+}\left(\mathbf{x}_i^{+}\right)\right)\left(\sum\limits_{\mathbf{u}^{t} \in \mathbb{D}_{\mathbf{U}^{t}}} \mathcal{C}^{-}\left(\mathbf{x}^{-}\right)\right)} B^{t}\left(\mathbf{m}^{t}\right) \tag{7.31}$$

$$= \alpha \sum_{\mathbf{m}^{t} \in \mathbb{D}_{\mathbf{M}^{t}}} \frac{\sum\limits_{\mathbf{u}^{t+1} \in \mathbb{D}_{\mathbf{U}^{t+1}}} \sum\limits_{\mathbf{u}^{t} \in \mathbb{D}_{\mathbf{U}^{t}}} \mathcal{C}^{+}\left(\mathbf{x}^{+}\right) \tau^{-}\left(\mathbf{x}^{-}\right)}{\sum\limits_{\mathbf{o}_i^{t+1} \in \mathbb{D}_{\mathbf{O}^{t+1}}} \sum\limits_{\mathbf{u}^{t+1} \in \mathbb{D}_{\mathbf{U}^{t+1}}} \sum\limits_{\mathbf{u}^{t} \in \mathbb{D}_{\mathbf{U}^{t}}} \mathcal{C}^{+}\left(\mathbf{x}_i^{+}\right) \mathcal{C}^{-}\left(\mathbf{x}^{-}\right)} B^{t}\left(\mathbf{m}^{t}\right) \tag{7.32}$$

Since

$$\begin{aligned}\mathcal{C}\left(\mathbf{m}^{t:t+1}, \mathbf{o}^{t:t+1}, \boldsymbol{\mu}^{t}, \mathbf{u}^{t:t+1}\right) &= \mathcal{C}^{+}\left(\mathbf{m}^{t+1}, \mathbf{o}^{t+1}, \mathbf{u}^{t+1}\right) \mathcal{C}^{-}\left(\mathbf{m}^{t:t+1}, \mathbf{o}^{t}, \mathbf{u}^{t}, \boldsymbol{\mu}^{t}\right) \\ &= \mathcal{C}^{+}\left(\mathbf{x}^{+}\right) \mathcal{C}^{-}\left(\mathbf{x}^{-}\right)\end{aligned}$$

If we let

$$\mathbf{x}^{t:t+1} = \mathbf{m}^{t:t+1} \cup \mathbf{o}^{t:t+1} \cup \boldsymbol{\mu}^{t} \cup \mathbf{u}^{t:t+1}$$
$$\mathbf{x}_i^{t:t+1} = \mathbf{m}^{t:t+1} \cup \mathbf{o}_i^{t+1} \cup \mathbf{o}^{t} \cup \boldsymbol{\mu}^{t} \cup \mathbf{u}^{t:t+1}$$

then re-writing Eq. 7.32 produces our final result, a recursive belief state update equation for PCCA:

$$B^{t+1}\left(\mathbf{m}^{t+1}\right) = \alpha \sum_{\mathbf{m}^t \in \mathbb{D}_{\mathbf{M}^t}} \frac{\sum_{\mathbf{u}^{t:t+1} \in \mathbb{D}_{\mathbf{u}^{t:t+1}}} \mathcal{C}\left(\mathbf{x}^{t:t+1}\right) \tau\left(\mathbf{x}^-\right)}{\sum_{\mathbf{o}_i^{t+1} \in \mathbb{D}_{\mathbf{O}^{t+1}}} \sum_{\mathbf{u}^{t:t+1} \in \mathbb{D}_{\mathbf{U}^{t:t+1}}} \mathcal{C}\left(\mathbf{x}_i^{t:t+1}\right)} B^t\left(\mathbf{m}^t\right) \tag{7.33}$$

More generally, we want to find the k most likely estimates, preferably without computing all of them:

$$\underset{\mathbf{m}^{t+1} \in \mathbb{D}_{\mathbf{M}^{t+1}}}{\arg^{\mathrm{k}}\max} \quad \mathbf{P}\left(\mathbf{m}^{t+1} | \mathbf{o}^{0:t+1}, \boldsymbol{\mu}^{0:t}\right) \tag{7.34}$$

We assume that when estimating the mode of our PCCA model that most of the probability density resides in a limited number of states. Thus, by determining the k most probable modes, we can approximate the distribution $\mathbf{P}\left(\mathbf{m}^{t+1} | \boldsymbol{\mu}^{0:t+1}, \mathbf{o}^{0:t+1}\right)$ in a way useful for planning and control purposes.

$$\begin{aligned} &\underset{\mathbf{m}^{t+1} \in \mathbb{D}_{\mathbf{M}^{t+1}}}{\arg^{\mathrm{k}}\max} \quad B^{t+1}\left(\mathbf{m}^{t+1}\right) \\ &\qquad = \alpha \underset{\mathbf{m}^{t+1} \in \mathbb{D}_{\mathbf{M}^{t+1}}}{\arg^{\mathrm{k}}\max} \sum_{\mathbf{m}^t \in \mathbb{D}_{\mathbf{M}^t}} \frac{\sum_{\mathbf{u}^{t:t+1} \in \mathbb{D}_{\mathbf{U}^{t:t+1}}} \mathcal{C}\left(\mathbf{x}^{t:t+1}\right) \tau\left(\mathbf{x}^-\right)}{\sum_{\mathbf{o}_i^{t+1} \in \mathbb{D}_{\mathbf{O}^{t+1}}} \sum_{\mathbf{u}^{t:t+1} \in \mathbb{D}_{\mathbf{U}^{t:t+1}}} \mathcal{C}\left(\mathbf{x}_i^{t:t+1}\right)} B^t\left(\mathbf{m}^t\right) \end{aligned} \tag{7.35}$$

And substituting our definition of τ from Eq. 7.19:

$$\underset{\mathbf{m}^{t+1} \in \mathbb{D}_{\mathbf{M}^{t+1}}}{\arg^{\mathrm{k}}\max} \sum_{\mathbf{m}^t \in \mathbb{D}_{\mathbf{M}^t}} \frac{\sum_{\mathbf{u}^{t:t+1} \in \mathbb{D}_{\mathbf{U}^{t:t+1}}} \mathcal{C}\left(\mathbf{x}^{t:t+1}\right) \prod_{m_a^{t:t+1} \in \mathbf{m}^{t:t+1}} \tau_a\left(\mathbf{x}^- \Downarrow_{\mathbf{X}_a^{t:t+1}}\right)}{\sum_{\mathbf{o}_i^{t+1} \in \mathbb{D}_{\mathbf{O}^{t+1}}} \sum_{\mathbf{u}^{t:t+1} \in \mathbb{D}_{\mathbf{U}^{t:t+1}}} \mathcal{C}\left(\mathbf{x}_i^{t:t+1}\right)} B^t\left(\mathbf{m}^t\right) \tag{7.36}$$

Thus computing the k most likely modes is a maximization over ratio of sum of products where the base terms are the transition probabilities from the model, the consistency of the assignment with the constraints, and the prior belief distribution. The key to encoding this equation is to realize that all of the terms are functions that map solutions of the constraints to constants: the transition

functions τ map the solutions to transition probabilities, the constraint function $\mathcal{C}$ maps the solutions to 1 and the non solutions to 0, and the belief function B^t maps solutions to the prior probability of previous mode. We use this insight in the next section to map this equation to a class of optimal constraint satisfaction problems.

7.4 Reduction of PCCA Belief State Estimation to an OCSP

Eq. 7.36 samples the k most probable states based on the previously estimated k most probable states. In order to solve Eq. 7.36, we employ the optimal constraint satisfaction problem (OCSP) solver of Chapter 5. The OCSP form specified in Chapter 5 is:

$$\underset{\mathbf{x}_M \in \mathbb{D}_{\mathbf{X}_M \setminus \mathbf{X}_a}}{\arg^{\mathrm{k}} \max} \sum_{\mathbf{x}_\Sigma \in \mathbb{D}_{\mathbf{X}_\Sigma \setminus \mathbf{X}_a}} \left(\prod_{i=1}^{m} g_i \left(\mathbf{x}_M \cup \mathbf{x}_\Sigma \cup \mathbf{x}_a \right) \cdot \prod_{i=m+1}^{n} \frac{1}{g_i \left(\mathbf{x}_M \cup \mathbf{x}_\Sigma \cup \mathbf{x}_a \right)} \cdot \alpha_{\mathbf{x}_\Sigma} \right) \tag{7.37}$$

Where g_i has the form:

$$g_i \left(\mathbf{x}_{M\Sigma a} \right) = \sum_{\mathbf{x}'_R \in \mathbb{D}_{\mathbf{X}'_R \subseteq \mathbf{X}_R}} \prod_{j=0}^{n_i} f_j \left(\left(\mathbf{x}_{M\Sigma a} \Downarrow_{\mathbf{X} \setminus \mathbf{X}'_R} \right) \cup \mathbf{x}'_R \right) \mathcal{C} \left(\left(\mathbf{x}_{M\Sigma a} \Downarrow_{\mathbf{X} \setminus \mathbf{X}'_R} \right) \cup \mathbf{x}'_R \right) \tag{7.38}$$

Eq. 7.36 can be encoded as an instance of the OCSP problem, through the instantiation:

$$\mathbf{X}_M = \mathbf{M}^{t+1} \qquad \mathbf{X}_a = \left\{ \mathbf{O}^{t+1}, \mathbf{C}^t, \mathbf{O}^t \right\}$$
$$\mathbf{X}_\Sigma = \mathbf{M}^t \qquad \alpha_{\mathbf{x}_\Sigma} = B \left(\mathbf{x}_\Sigma \right)$$

$$\underset{\mathbf{m}^{t+1} \in \mathbb{D}_{\mathbf{M}^{t+1}}}{\arg^{\mathrm{k}} \max} \sum_{\mathbf{m}^t \in \mathbb{D}_{\mathbf{M}^t}} \frac{g_1 \left(\mathbf{m}^{t:t+1} \cup \mathbf{o}^{t:t+1} \cup \boldsymbol{\mu}^t \right)}{g_2 \left(\mathbf{m}^{t:t+1} \cup \mathbf{o}^{t:t+1} \cup \boldsymbol{\mu}^t \right)} \cdot B \left(\mathbf{m}^t \right)$$

$$
\begin{aligned}
\mathbf{x}_{M\Sigma a} &= \mathbf{m}^{t:t+1} \cup \mathbf{o}^{t:t+1} \cup \boldsymbol{\mu}^{t} \\
g_1\left(\mathbf{x}_{M\Sigma a}\right) &= \sum_{\mathbf{u}^{t:t+1} \in \mathbb{D}_{\mathbf{U}^{t:t+1}}} \prod_{a} f_a\left(\mathbf{x}_{M\Sigma a} \cup \mathbf{u}^{t:t+1}\right) \mathcal{C}\left(\mathbf{x}_{M\Sigma a} \cup \mathbf{u}^{t:t+1}\right) \\
f_a\left(\mathbf{x}^{t:t+1}\right) &= \tau_a\left(\mathbf{x} \Downarrow_{\mathbf{X}_a^{t:t+1}}\right) \\
g_2\left(\mathbf{x}_{M\Sigma a}\right) &= \sum_{\mathbf{o}_i^{t+1}, \mathbf{u}^{t:t+1} \in \mathbb{D}_{\mathbf{O}_i^{t+1}, \mathbf{U}^{t:t+1}}} \mathcal{C}\left(\mathbf{m}^{t:t+1} \cup \mathbf{o}^{t} \cup \boldsymbol{\mu}^{t} \cup \mathbf{o}_i^{t+1} \cup \mathbf{u}^{t:t+1}\right)
\end{aligned}
$$

We note that the numerator and denominator filter out any constraints that only apply to time t, given $\mathbf{m}^t$ and $\boldsymbol{\mu}^t$. We can thus simplify the estimation problem to only include variables and constraints relevant to deciding the transition constraints τ_a. In particular, if all transitions depend only on commands, then all of the time t constraints can be eliminated, along with all variables at time t, except $\mathbf{m}^t$ and $\boldsymbol{\mu}^t$.

As another improvement, we conjecture that the size of the representation generated by the OCSP solver can be reduced by estimating the value of more variables. More specifically, since the decomposition algorithm used by the OCSP solver must first partition the constraints using estimated variables before considering other variables, in order to ensure the maximization is correct, estimating the value of more variables should lead to a better decomposition by giving the decomposition algorithm more flexibility. The decomposition can use this flexibility to break the problem into a larger set of smaller pieces. Estimating the value of more variables adds their time $t+1$ version to $\mathbf{X}_M$ and their time t counterparts to $\mathbf{X}_\Sigma$. The limitation of this approach is that the k estimates may distinguish between values of these new variables, for which we presumably do not care to distinguish, as we were not estimating them. None the less, we anticipate that, in some circumstances, estimating some variables from $\mathbf{u}$ produces a decrease in the size of the compiled representation that is sufficient to make up for the increased k required to get comparable results.

7.5 Approximating the PCCA Observation Distribution

In Chapter 4, we are interested in estimating the probability $\mathbf{P}\left(\mathbf{o}^{t+1}|B\right)$, where B is the predicted belief state computed by Eq. 7.36 given no observations. We build upon Eq. 7.23:

$$\mathbf{P}\left(\mathbf{o}^{t+1}|\mathbf{m}^{t+1}\right) = \frac{\sum\limits_{\mathbf{u}^{t+1}\in\mathbb{D}_{\mathbf{U}^{t+1}}} \mathcal{C}^{+}\left(\mathbf{m}^{t+1}, \mathbf{o}^{t+1}, \mathbf{u}^{t+1}\right)}{\sum\limits_{\mathbf{o}_i^{t+1}\in\mathbb{D}_{\mathbf{O}^{t+1}}} \sum\limits_{\mathbf{u}^{t+1}\in\mathbb{D}_{\mathbf{U}^{t+1}}} \mathcal{C}^{+}\left(\mathbf{m}^{t+1}, \mathbf{o}_i^{t+1}, \mathbf{u}^{t+1}\right)},$$

which says that the probability of an observation is the number of solutions that are consistent with the observation over the number of solutions that are consistent with any observation. We can naturally extend this to $\mathbf{P}\left(\mathbf{o}^{t+1}|B\right)$:

$$\begin{aligned}
\mathbf{P}\left(\mathbf{o}^{t+1}|B\right) &= \sum_{\mathbf{m}^{t+1}\in\mathbb{D}_{\mathbf{M}^{t+1}}} \mathbf{P}\left(\mathbf{o}^{t+1}, \mathbf{m}^{t+1}|B\right) && (7.39)\\
&= \sum_{\mathbf{m}^{t+1}\in\mathbb{D}_{\mathbf{M}^{t+1}}} \mathbf{P}\left(\mathbf{o}^{t+1}|\mathbf{m}^{t+1}\right)\mathbf{P}\left(\mathbf{m}^{t+1}|B\right) && (7.40)\\
&= \sum_{\mathbf{m}^{t+1}\in\mathbb{D}_{\mathbf{M}^{t+1}}} \mathbf{P}\left(\mathbf{o}^{t+1}|\mathbf{m}^{t+1}\right) B\left(\mathbf{m}^{t+1}\right) && (7.41)\\
&= \sum_{\mathbf{m}^{t+1}\in\mathbb{D}_{\mathbf{M}^{t+1}}} \frac{\sum\limits_{\mathbf{u}^{t+1}\in\mathbb{D}_{\mathbf{U}^{t+1}}} \mathcal{C}^{+}\left(\mathbf{m}^{t+1}, \mathbf{o}^{t+1}, \mathbf{u}^{t+1}\right)}{\sum\limits_{\mathbf{o}_i^{t+1}\in\mathbb{D}_{\mathbf{O}^{t+1}}} \sum\limits_{\mathbf{u}^{t+1}\in\mathbb{D}_{\mathbf{U}^{t+1}}} \mathcal{C}^{+}\left(\mathbf{m}^{t+1}, \mathbf{o}_i^{t+1}, \mathbf{u}^{t+1}\right)} B\left(\mathbf{m}^{t+1}\right) && (7.42)
\end{aligned}$$

As with belief state estimation, if we are interested in only computing the k most probable observations, then we can augment Eq. 7.42 by adding an $\arg^{\mathrm{k}}\max$:

$$\arg^{\mathrm{k}}\max_{\mathbf{o}^{t+1}\in\mathbb{D}_{\mathbf{O}^{t+1}}} \sum_{\mathbf{m}^{t+1}\in\mathbb{D}_{\mathbf{M}^{t+1}}} \frac{\sum\limits_{\mathbf{u}^{t+1}\in\mathbb{D}_{\mathbf{U}^{t+1}}} \mathcal{C}^{+}\left(\mathbf{m}^{t+1}, \mathbf{o}^{t+1}, \mathbf{u}^{t+1}\right)}{\sum\limits_{\mathbf{o}_i^{t+1}\in\mathbb{D}_{\mathbf{O}^{t+1}}} \sum\limits_{\mathbf{u}^{t+1}\in\mathbb{D}_{\mathbf{U}^{t+1}}} \mathcal{C}^{+}\left(\mathbf{m}^{t+1}, \mathbf{o}_i^{t+1}, \mathbf{u}^{t+1}\right)} B\left(\mathbf{m}^{t+1}\right) \qquad (7.43)$$

Eq. 7.43 lets us sample the k most probable observations and is nearly in the form required by the OCSP solver of Chapter 5. The only part that is not of the correct form is the denominator; the denominator sums over $\mathbf{o}_i^{t+1}$, which is not part of $\mathbf{X}_R$, the variables that we can sum over in g_i. It is also only a function of $\mathbf{m}^{t+1}$. We can thus compile it separately from the rest of the problem, using the techniques for compiling g^* in Chapter 5. We can then evaluate this part given each $\mathbf{m}^{t+1}$ and incorporate it with $B\left(\mathbf{m}^{t+1}\right)$ directly.

The optimal constraint satisfaction problem form required by Chapter 5 is:

$$\underset{\mathbf{x}_M \in \mathbb{D}_{\mathbf{X}_M \setminus \mathbf{X}_a}}{\arg^{\mathrm{k}}\max} \sum_{\mathbf{x}_\Sigma \in \mathbb{D}_{\mathbf{X}_\Sigma \setminus \mathbf{X}_a}} \left(\prod_{i=1}^{m} g_i\left(\mathbf{x}_M \cup \mathbf{x}_\Sigma \cup \mathbf{x}_a\right) \cdot \prod_{i=m+1}^{n} \frac{1}{g_i\left(\mathbf{x}_M \cup \mathbf{x}_\Sigma \cup \mathbf{x}_a\right)} \cdot \alpha_{\mathbf{x}_\Sigma} \right) \tag{7.44}$$

Where g_i has the form:

$$g_i\left(\mathbf{x}_{M\Sigma a}\right) = \sum_{\mathbf{x}'_R \in \mathbb{D}_{\mathbf{X}'_R \subseteq \mathbf{X}_R}} \prod_{j=0}^{n_i} f_j\left(\left(\mathbf{x}_{M\Sigma a} \Downarrow_{\mathbf{X} \setminus \mathbf{X}'_R}\right) \cup \mathbf{x}'_R\right) \mathcal{C}\left(\left(\mathbf{x}_{M\Sigma a} \Downarrow_{\mathbf{X} \setminus \mathbf{X}'_R}\right) \cup \mathbf{x}'_R\right) \tag{7.45}$$

Eq. 7.43 can be encoded as an instance of the OCSP problem with the instantiation:

$$\begin{aligned} \mathbf{X}_M &= \mathbf{O}^{t+1} & \mathbf{X}_a &= \{\} \\ \mathbf{X}_\Sigma &= \mathbf{M}^{t+1} & \alpha_{\mathbf{x}_\Sigma} &= \tfrac{B(\mathbf{x}_\Sigma)}{g_2(\mathbf{x}_\Sigma)} \end{aligned}$$

$$\underset{\mathbf{o}^{t+1} \in \mathbb{D}_{\mathbf{O}^{t+1}}}{\arg^{\mathrm{k}}\max} \sum_{\mathbf{m}^{t+1} \in \mathbb{D}_{\mathbf{M}^{t+1}}} g_1\left(\mathbf{o}^{t+1} \cup \mathbf{m}^{t+1}\right) \cdot \frac{B\left(\mathbf{m}^{t+1}\right)}{g_2\left(\mathbf{m}^{t+1}\right)}$$

$$\begin{aligned} g_1\left(\mathbf{o}^{t+1} \cup \mathbf{m}^{t+1}\right) &= \sum_{\mathbf{u}^{t+1} \in \mathbb{D}_{\mathbf{U}^{t+1}}} \mathcal{C}\left(\mathbf{o}^{t+1} \cup \mathbf{m}^{t+1} \cup \mathbf{u}^{t+1}\right) \\ g_2\left(\mathbf{m}^{t+1}\right) &= \sum_{\mathbf{o}^{t+1}, \mathbf{u}^{t+1} \in \mathbb{D}_{\mathbf{O}^{t+1}, \mathbf{U}^{t+1}}} \mathcal{C}\left(\mathbf{m}^{t+1} \cup \mathbf{o}^{t+1} \cup \mathbf{u}^{t+1}\right) \end{aligned}$$

7.6 Conclusion

This chapter began by reviewing the update equations for a Bayesian Filter given an observation sequence. We then reviewed the PCCA model as a connected set of discrete components. Next, we derived exact and approximate equations for PCCA estimation under the assumption of uniform likelihood.

Finally, using this derivation, we then reformulated the approximate PCCA update as an OCSP as defined in Chapter 5 as well as reformulated the observation probability sub-problem as its own instance of an OCSP.

Chapter 8

Results and Conclusion

This chapter presents some results of running the task monitoring capability presented in this thesis on two examples followed by a discussion on future work. This chapter then concludes this thesis.

8.1 Results

The results presented here are generated using a C++ implementation of the algorithms. We use Stedl [52] to execute the dispatchable plans. We label the plan with partial states, and thus require a planner to interpret the correct command to achieve these partial states. We use the Model-based Reactive Planner [8] for this purpose. These test results are generated on a 2.4 GHz Pentium® 4 with 2 GB of RAM running Windows® XP.

The compilation specified in Chapter 5 was implemented using the C2D sd-DNNF compiler [19]. The compiler is implemented using binary variables, and so the resulting sd-DNNF is also represented with binary variables. We convert the sd-DNNF into an equivalent sd-DNNF with multi-valued variables. After this conversion, and for each x multi-valued assignment in the sd-DNNF, if x is constant for any possible observation with respect to $f_{\mathbf{x}_{M\Sigma a}}(x)$, then the compilation step replaced x with the constant. This substitution of the value for the assignment marginally improves online evaluation by itself, as it reduces the number of terms in $f_{\mathbf{x}_{M\Sigma a}}$. This change allows for the further reduction of the size of the sd-DNNF by combining like terms, though this implementation

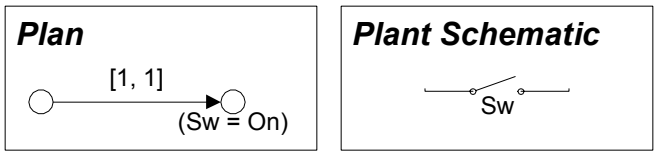

Figure 8-1: This figure shows the simple switch on the right and the sample plan used in this chapter to generate results on the left. The switch is modeled with only three states: on, off, and broken. The plan requests that the switch be on at time 1.

does not take advantage of this additional optimization step.

The tests sample if the plan will succeed, a Boolean value, and record the number of true results divided by the number of samples iteratively. For example, the sequence (true, false, true) is recorded as (1, 0.5, 0.6667). In Fig. 8-3, this recorded value is plotted for a single recording of the switch example. The remaining plots print the statistics of each recorded value based on multiple runs of the same length.

The results in this section are based on two examples. The first example is a simple switch that is being commanded to turn on, shown in Fig. 8-1. The second example is a propulsion system that is controlled through an external controller, shown in Fig. 8-2. The propulsion system is being commanded into standby mode, in our case from fully off.

8.1.1 Switch Example

For the switch example it is possible to track all three states, so this example is able to demonstrate that the probability that the plan will succeed converges to the true probability of 59.8%. In this example, the initial belief is that the switch is failed with a probability of 40% and is off with a probability of 60%. The plan requires the switch to be in the on state in one step, which is only possible from the off state. There is a 0.2% chance (based on a 0.3% failure rate from off to on and a 60% chance of being off) that the switch does not make it from off to on but instead fails.

For the switch example, each sample took 4.01ms to generate with a standard deviation of 1.04ms. Recall that the online algorithm requires a compiled sd-DNNF for three different steps. The first step is the estimation algorithm that computes the belief state update equations. This

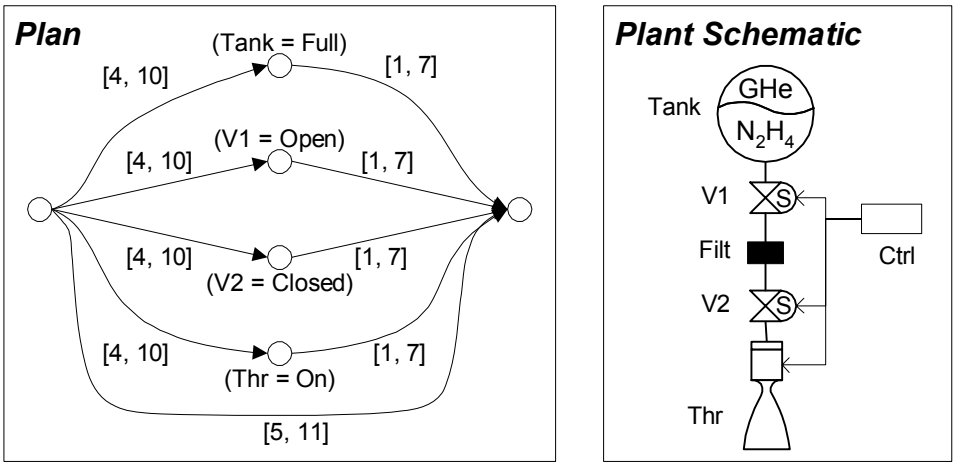

Figure 8-2: This figure shows the propulsion system example on the right and the sample plan used in this chapter to generate results on the left. The propulsion system consists of a tank, a pair of valves, a fuel filter, a thruster, and a controller. The propulsion system has a number of recovery mechanisms in the event of failure such as resetting the controller.
The plan requests that the propulsion system be put into standby mode in 5-11 steps. Standby mode consists in ensuring sufficient fuel exists, turning on the first and off the second valves, and turning the thruster power on. All valve and thruster commands must go through the controller, which must thus also be configured to be in an appropriate state to allow for the commanding. The plan does not require that any of these happen in any particular order.

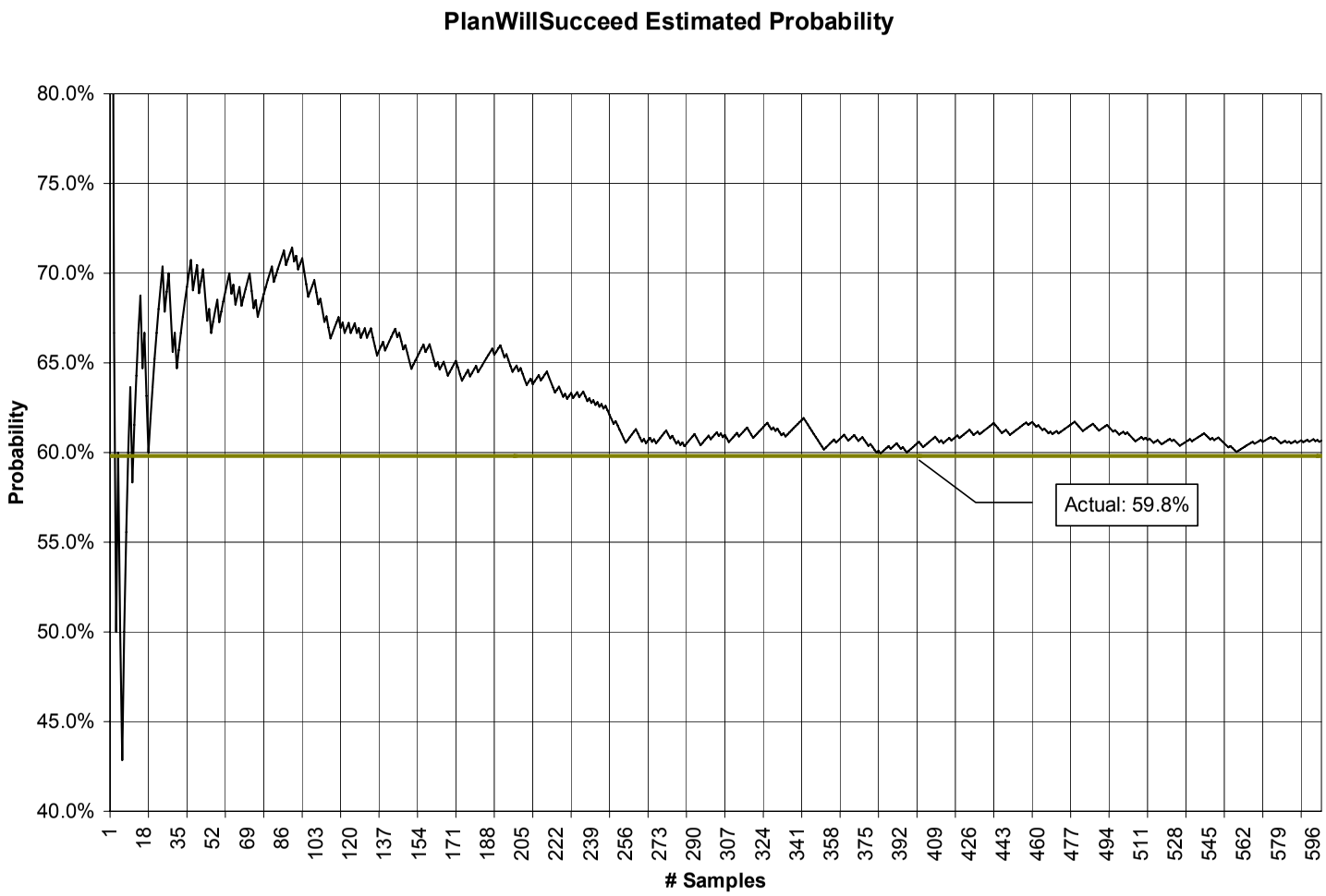

Figure 8-3: This figure depicts the anytime probability that the plan will succeed for the switch example. As the number of samples reaches 600 the probability is within converging to the actual probability of 59.8%, with an error just under 1% after 600 samples.

estimation sd-DNNF has 38 nodes and 48 arcs for this switch example. The second step is the observation sampling function. The observation sd-DNNF consists of two parts, one to compute the reciprocal weighting of each state's probability and then the normal observation sd-DNNF used to sample the weighted observation. The observation weighting sd-DNNF has 10 nodes and 10 arcs. The main observation sd-DNNF has 11 nodes and 12 arcs. The final step involves computing the probability of transitioning from a prior state to the next state, ignoring observations, and is used to update the previous belief state probabilities prior to sampling a trajectory. The transition probability's sd-DNNF has 37 nodes and 49 arcs. These node and arc sizes are listed in Table 8.1.

8.1.2 Propulsion Example

For this propulsion example, we track only the top 10 states, discarding all others at each step. This example has 2,025 possible states, which is to say that the HMM representation of example would have a matrix with 4 million entries. The example had the initial belief state that the system was fully off with 100% probability. The plan requests that the propulsion system be put into standby mode, which means that all the devices are on except the second valve, which stays closed. The probability that the plan will succeed in this case converges to about 93.46% with a standard deviation of 1.31% at 400 samples, as shown in Fig. 8-6. The sampled execution of the plan has a median length of 6 steps. The failure rate is consistent with the modeled high rate of failure of the tank and the valves. For this example, each sample took 89ms to generate with a standard deviation of 12ms.

We ran the same example, but instead tracking 30 states (k=30) and the results are shown in Fig. 8-7. With the additional precision of tracking more states, the probability that the plan will succeed converges to about the same probability – 93.50% with a standard deviation of 1.29% at 400 samples. Tracking 30 states instead of 10 states required 203ms to generate a sample with a standard deviation of 25ms.

The estimation sd-DNNF has 2,123 nodes and 9,092 arcs. The observation weighting sd-DNNF has 148 nodes and 224 arcs. The main observation sd-DNNF has 503 nodes and 969 arcs. The transition probability's sd-DNNF has 880 nodes and 1,824 arcs. These are shown along with the switch quantities in Table 8.1.

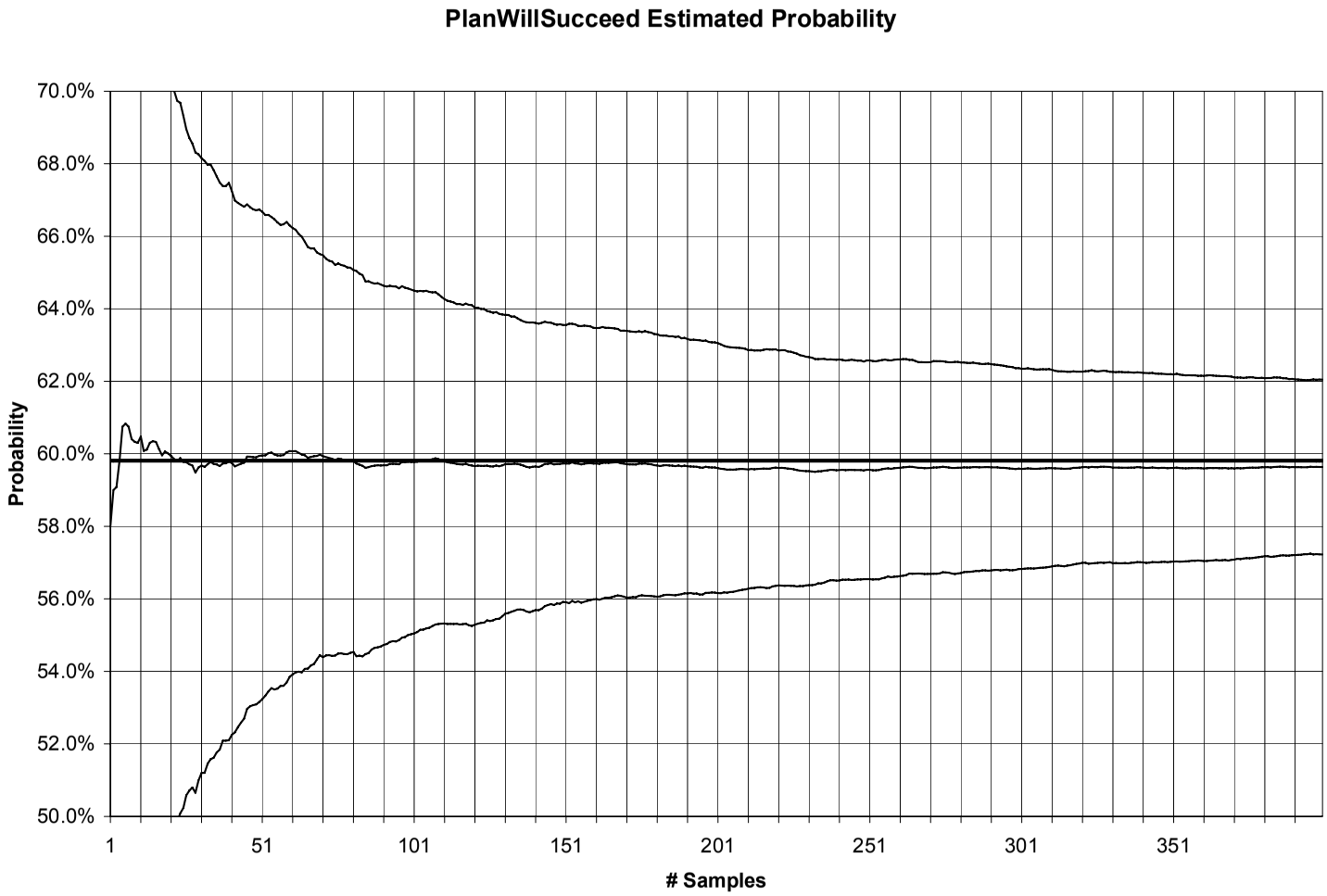

Figure 8-4: This figure depicts the statistics of the anytime probability that the plan will succeed for the switch example. The top and bottom curves depict one standard deviation from the mean of 400 different runs of the anytime algorithm with the center line depicting the stable mean of the 400 runs. At 400 samples, the standard deviation is a little less than 2.4%.

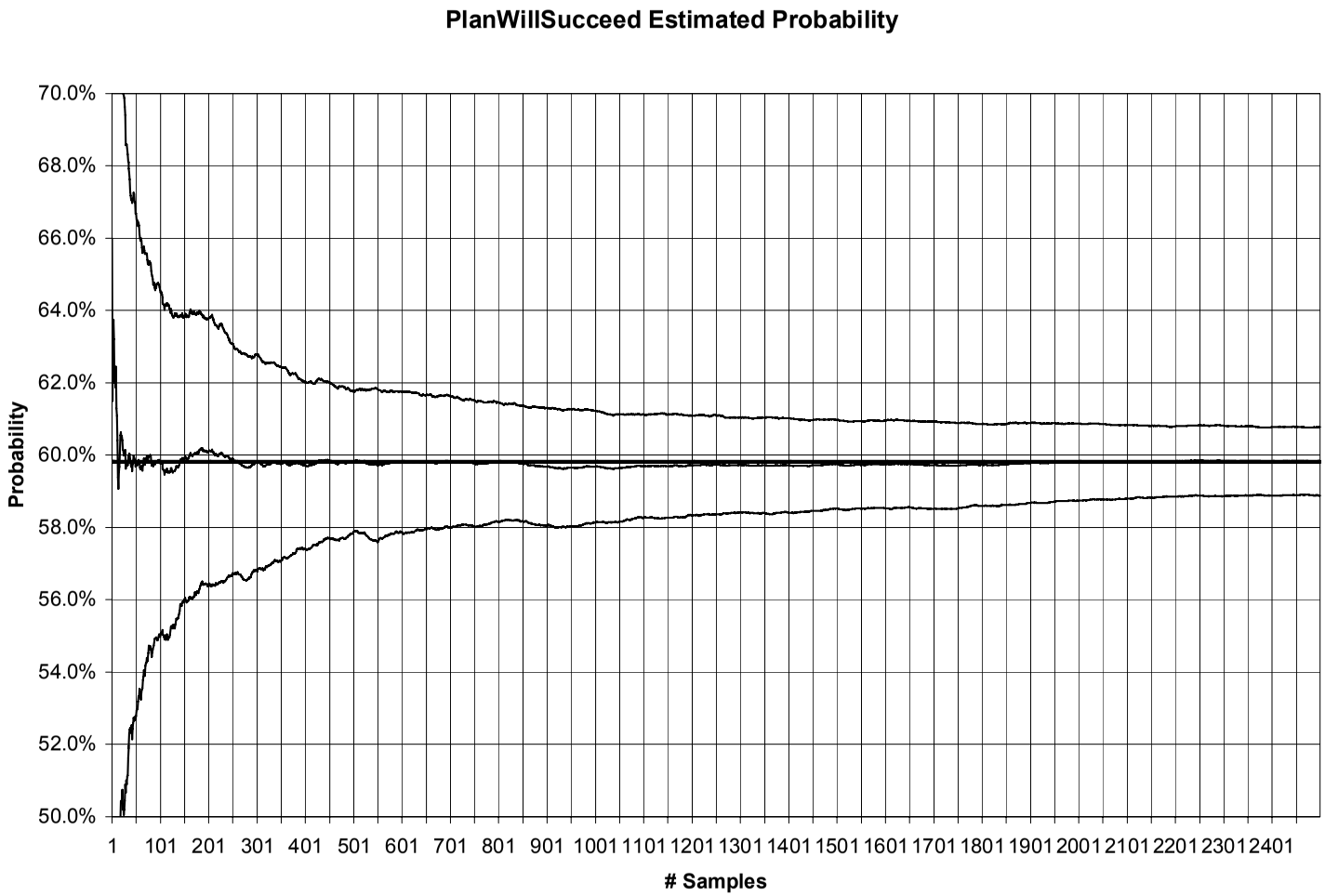

Figure 8-5: This figure depicts the statistics of the anytime probability that the plan will succeed for the switch example. The top and bottom curves depict the standard deviation of 100 different runs of the anytime algorithm with the center line depicting the mean of the 100 runs. At 2500 samples, the standard deviation is a little less than 1%.

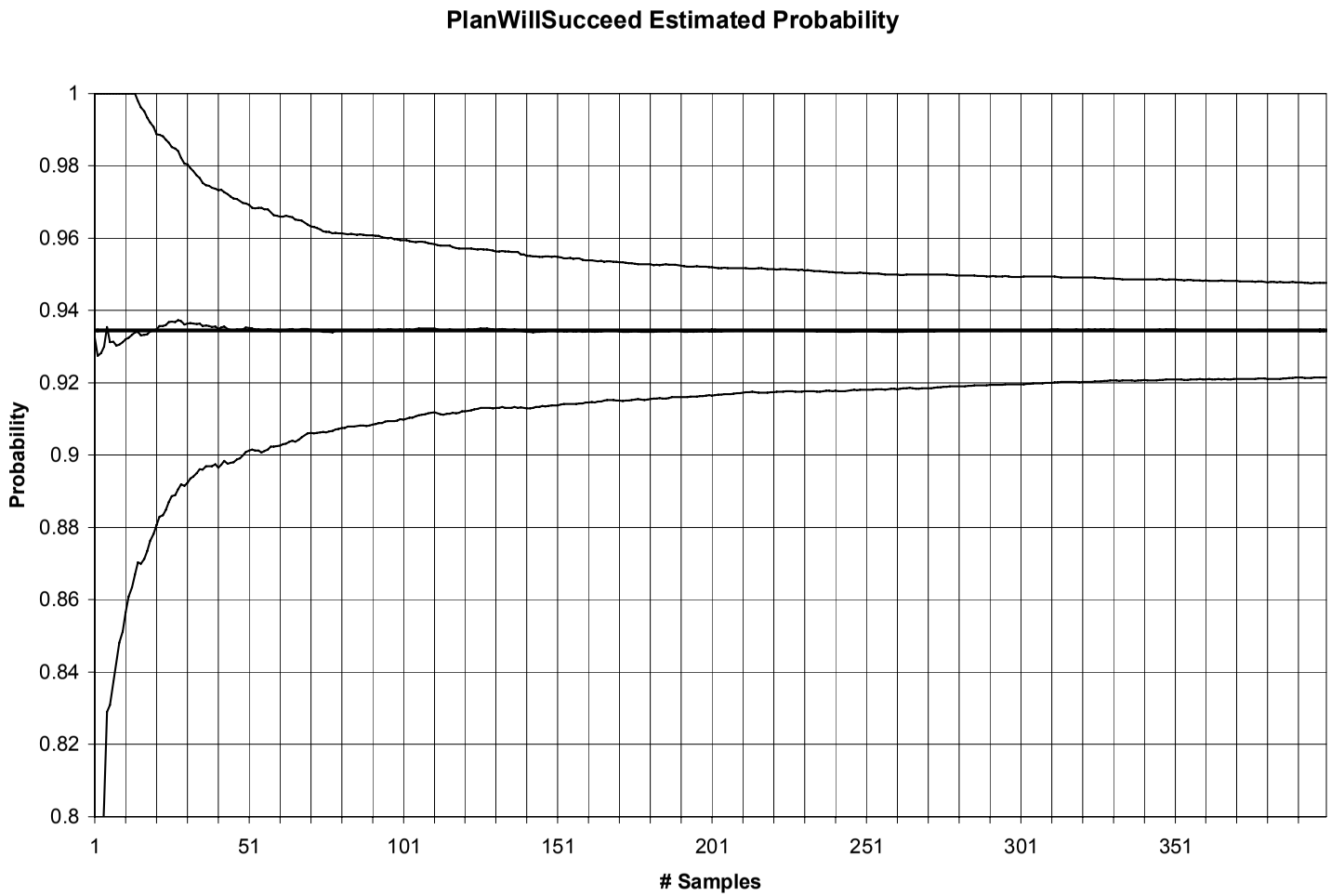

Figure 8-6: This figure depicts the statistics of the anytime probability that the plan will succeed for the propulsion example. The top and bottom curves depict the standard deviation of 400 different runs of the anytime algorithm with the center line depicts the mean of the 400 runs. At 400 samples, the standard deviation is about 1.3%.

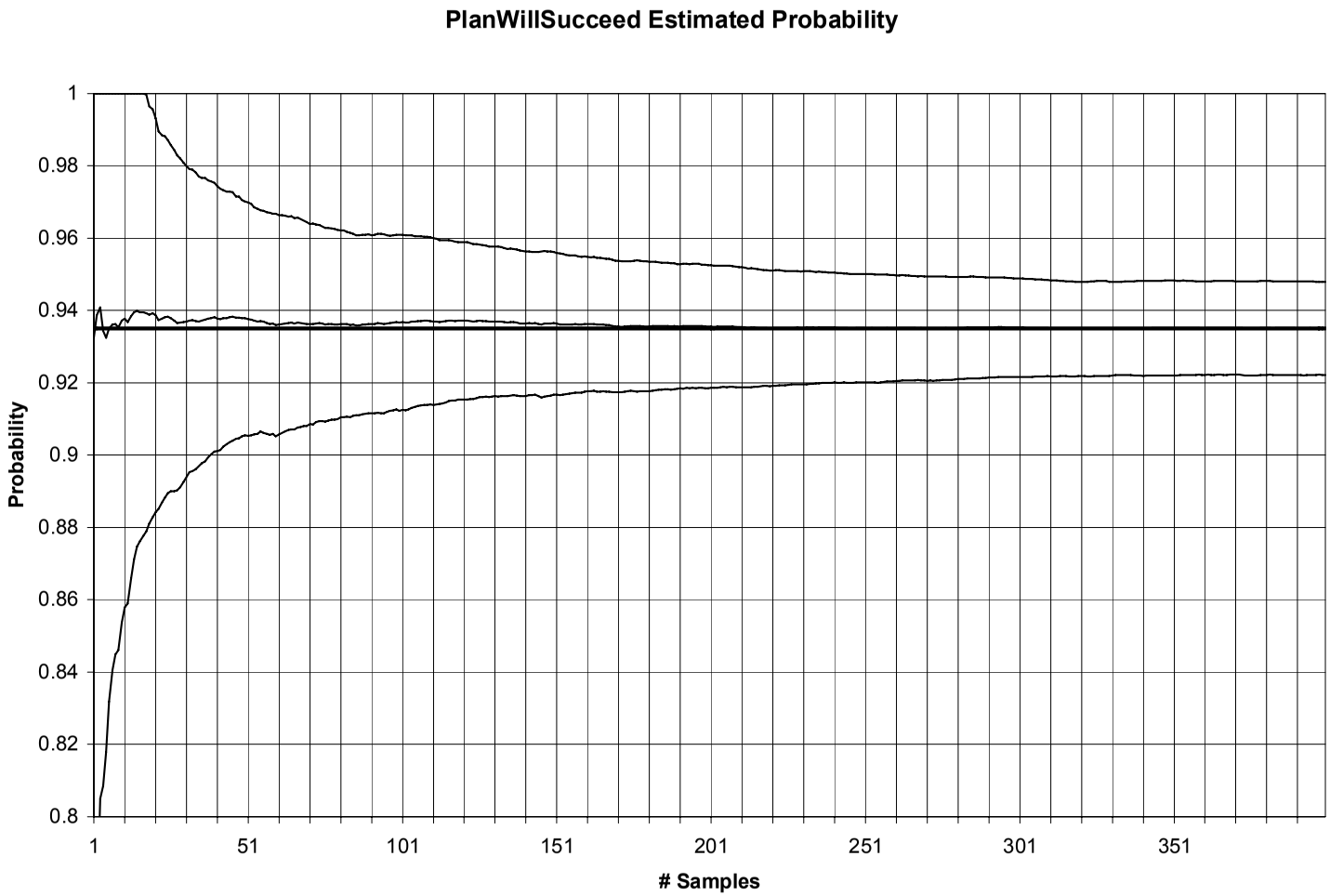

Figure 8-7: This figure depicts the statistics of the anytime probability that the plan will succeed for the propulsion example, when we let k=30 instead of k=10. The top and bottom curves depict the standard deviation of 400 different runs of the anytime algorithm with the center line depicts the mean of the 400 runs. At 400 samples, the standard deviation is about 1.3%.

	$Estimation$		$ObsDenom$		$ObsSample$		$Transition$	
	$Nodes$	$Arcs$	$Nodes$	$Arcs$	$Nodes$	$Arcs$	$Nodes$	$Arcs$
$Switch$	38	48	10	10	11	12	37	49
$Propulsion$	$2,123$	$9,092$	148	224	503	969	880	$1,824$

Table 8.1: These tables show the sizes of the four sd-DNNFs used to compute the probability that the plan will succeed. These are used in three different steps of the probability computation: the belief state estimation, the observation sampling function (with two parts), and the transition probability computation.

Both sets of these results show that the amount of time taken to sample a single potential execution is small, on the order of what one would expect to see for a system that typically runs at around 1 Hz on an embedded processor. The propulsion example is six simulation steps per estimate, on average, so only about 15ms per step. The rate of convergence for the probability distribution is also as expected, specifically the predicted standard deviation at 400 samples is less than 5%, and for the switch example it was about 2.4% and for the other only 1.3%. At 2,500 samples the expected error is less than 2% and for the switch it was about 1%.

With respect to both sets of results, it is worth noting that the implementation is an amalgam of a number of research implementations, produced by several parties, and thus a non-insignificant amount of time is spent translating results between the different modules that comprise the plan monitoring capability presented in this thesis. Thus, it is expected that the results presented here can be substantially improved upon through improvements to the software design. In addition, several tools are not designed to be re-run repeatedly to generate samples and thus needed to be re-initialized for every sample, rather than being able to save and restore the runtime structure directly.

The second propulsion example took about 22x longer to generate a single sample as compared with the switch example, which is consistent with its larger size and longer plan length. It has around 38x more nodes and 102x more arcs than the simple switch example and 6x the length. This is evidence that there is a significant constant-time component of the 4ms of the switch example, as otherwise the propulsion example would have taken some 600x longer (arcs $\times$ plan length). The algorithm and implementation also have substantial room for improvement. 89ms per sample is

still rather long considering the samples are generated on a relatively fast machine – this propulsion example is not yet truly complex and it already takes 35 seconds to generate an estimate with a 1.3% standard deviation (400 samples). Fortunately there appears to be numerous avenues for further improvement to the algorithm, as described in the next section, beyond just improving the implementation.

It is also worth noting that the propulsion system's model has about 3.5x more arcs than states in the system. This at present seems more likely than not given the current methodology used to generate the constraints fed to the C2D compiler. The additional decomposition specified in Chapter 5 are possible in a full implementation for the purpose of generating the estimation sd-DNNF, but these extra techniques require substantially altering C2D. This avenue was not pursued as the source code of C2D was not readily available and re-implementing C2D was beyond the scope of this thesis. With such a change, the Estimation sd-DNNF would be expected to be no larger than the present transition sd-DNNF, or about a quarter the size (slightly smaller than the number of states). In general this reduction is expected to require adding a search component to the online algorithm, as explained below.

The last three columns of Table 8.1 are likely compactible due to the introduction of values instead of assignments, but otherwise reflective of this thesis's technique. It should be noted that the propulsion example is a moderately coupled system, so it is not surprising that the resulting compiled model is a bit larger. Less coupled systems should generate substantially smaller sd-DNNFs as compared to the number of states of the system. The next section explains possible improvements to this work.

8.2 Future Work

8.2.1 sd-DNNF compression

A simple and effective way to reduce the runtime would be to take a second pass at the sd-DNNF, after replacing assignment labels with values, and re-compress the sd-DNNF. The C2D tool does this for the initial version of the sd-DNNF, but further compression is possible once assignments are

	$Estimation$		$ObsDenom$		$ObsSample$		$Transition$	
	$Total$	$Valued$	$Total$	$Valued$	$Total$	$Valued$	$Total$	$Valued$
$Switch$	18	7	5	2	5	0	18	9
$Propulsion$	129	59	51	32	51	24	129	79

Table 8.2: This table shows the number of leaf nodes that are set to pure values in the sd-DNNF. This gives a notion of the number of nodes that can be combined by applying the memoization technique suggested in this section. Further reduction should be possible by pre-computing the addition and multiplication of internal nodes with 2 or more leaves that are just pure values.

no longer distinguished, by explicitly combining them with $+$ and $\times$ as well as memoizing equal values. This reduces the overall number of nodes and arcs in the graph and thus also reduces the online complexity proportionately.

The current implementation, as specified above, replaces assignments with values, such as a transition assignment with its probability, say 99%. Distinguishing between different transitions is important for ensuring the solutions generated are properly assigned probabilities but is unimportant once the sd-DNNF is constructed, as it explicitly contains each solution once. By memoizing all transition assignments with the 99% probability into a single leaf, one can start to take advantage of additional memoization in the graph, and where possible, one can pre-compute constant additions and multiplications. For example, if an **And** node a has two children, m and n, that are leaves with values, then it can create a new leaf child v whose value is $\mathcal{L}_{\mathbf{P}}(m) \times \mathcal{L}_{\mathbf{P}}(n)$. This pre-computes the multiplication that would otherwise be done online for every evaluation of g^*.

The number of leaves that are set to probabilities as well as the total number of leaves for both examples is shown in Table 8.2. It will come as no surprise that the Estimation and Transition sd-DNNFs are based on both the previous and next set of variables, along with the transition probabilities, and thus have a bit over twice the number of assignments and leaves as compared to the observation function. For most of these sd-DNNFs, over a third of all leaves are set to pure values, and about half of those leaves set to values are set to the value 1. This introduces a substantial opportunity for improvement.

8.2.2 Parallel Sampling

When generating samples, the current algorithm generates each sample sequentially in a single thread. The growing abundance of multi-core processors suggests that some parallelism in the generation of samples would be beneficial. Since each sample is completely independent and we need hundreds of samples, an effective parallel algorithm is easily obtained by just computing samples in parallel.

8.2.3 Improved Sampling Techniques

There is a rich body of research in sampling techniques. This thesis uses a very basic sampling technique that works but we sample a complete set of observations and a plausible trajectory for every sample. It is possible to trade-off of the breadth of possibilities covered by the samples when using multiple completely independent samples with the computational advantage of re-using parts of previous samples to generate additional samples, reducing the overall computation required. This trade-off is between computing a better estimate of a particular partial sample's probability of success, which is necessarily cheaper than a full sample because part of the sampling has already been computed, and considering more near-term events, which are more expensive but reduces the risk that we will accidentally focus to much on an unrepresentative sequence. These techniques are applied, for instance, for sampling approaches to ray-tracing in computer graphics.

8.2.4 Search

We have favored search-free approaches to estimation both of the PCCA models and of the probability that the plan will succeed. This is in contrast to prior work in estimation such as [42], which relied primarily on search techniques to generate estimates. It is likely that work on hybrid search/explicit graph approaches can have significant benefit here as there are some cases where the generated sd-DNNF does not contain ample memoization and in such situations, decomposition can be done online inside of a search algorithm with similar efficiency. The decomposition generated by C2D is likely a good starting place for future work. A natural boundary for our problem is the boundary in the sd-DNNF created by those **And** nodes that split the $+$ nodes from the $\max$ nodes.

It is expected that introducing a search routine within the $\max$ nodes part of the graph will lead to a faster algorithm with less memory overhead.

Additionally, since a number of subgraphs of the sd-DNNF can be zero for any particular evaluation of the graph, the hybrid-search approach has the potential for savings realized by not needing to evaluate part of the graph by detecting zero subgraphs.

Improved Enumeration Algorithm

The k-best-solutions algorithm in Chapter 6 works by pre-computing a substantial amount of data and pushing that up to the root node. In general, most of the information generated is not needed to generate the final result. A search-based approach of computing only what's needed to generate the solution at the root such as the one used by Sachenbacher [49] should prove substantially faster.

Alternate Enumeration Algorithm

The k-best-solutions algorithm in Chapter 6 is assuming it is solving a maximization of products, which means the summations need to all be pre-computed: $\arg^{\mathrm{k}} \max_{\mathbf{x}_{M\setminus a} \in \mathbb{D}_{\mathbf{X}_M \setminus \mathbf{X}_a}}$. In practice this means that the maximization cannot be partitioned for most problems with $k > 1$. Otherwise, the partitioned maximization "forgets" which prior belief led to the next belief, and that can allow contradictory combinations of the two maximizations. If the maximization is not factored, then $k > 1$ poses no problem, but then we need to represent all the states, which we already stated previously was intractable. If we instead modify the enumeration algorithm to solve: $\arg^{\mathrm{k}} \max_{\mathbf{x}_{M\setminus a} \in \mathbb{D}_{\mathbf{X}_M \setminus \mathbf{X}_a}} \sum_{\mathbf{x}_{\Sigma\setminus a} \in \mathbb{D}_{\mathbf{X}_\Sigma \setminus \mathbf{X}_a}}$, then we can safely decompose the maximization and use the more complex enumeration algorithm to extract the k-best-solutions, while making sure the solutions have consistent support from each previous belief state. This type algorithm will need to rely on the prior sections change to a search-based approach as the maximum number of solutions needed at any maximal-product node is no longer bounded by k, and will thus need to scale to as many as is needed. Intuitively, by keeping the summation in this part of the problem, we have forced the products to remain inside the summation, so the best answer at a maximal-product node is going to be the sum of the product of the best answers of each child for each of the k belief states. If the

best answer for one child depends on a different previous belief than the best answer for another child, then the product of those two best options is a comparatively small value.

8.2.5 Observation Probabilities

Presently observation probabilities are computed using counting, which makes the equations very clean for the purpose of this thesis, though it is a potentially cumbersome mechanism for specifying the observation probability function in some circumstances. One possible direction for future work is to incorporate an extra observation weight into the probability computation. This could take the form of a set of f_j in the equations, much like the current transitions. The current implementation saves a substantial amount of overhead in the compilation phase by not needing to worry about the difference between unknown and observable variables with respect to their two summations. Introducing an observation term that *replaces* the existing computation will make compilation substantially more complicated (if not impossible), but just adding an extra weight into the computation should be easy to implement. A weight will necessarily have more complex semantics.

8.2.6 Simplified Observation Function

Instead of improving the observation probability specification, another option would be to actually return to Martin's probability specification, which is substantially easier to compute. It should be easier to generate an appropriate sd-DNNF from Martin's PCCA estimation equations and should also generate a substantially smaller graph. Tests of the estimation specific part of this thesis have suggested that for at least some examples, this thesis can take almost 10x longer to generate an estimate than Martin's algorithm, though this thesis is using more accurate equations. Some of the techniques used in this thesis related to compiling probabilistic constraints should carry over to Martin's work directly and it's possible that Martin's less complex computation can be fully encoded in the compiled data structure used in this thesis with some additional work.

8.3 Conclusion

This thesis has presented a novel algorithm for computing the probability of plan success using an explicit, probabilistic model of the physical plant. This plan monitoring capability allows for the closed-loop execution of plans, and in general the plan's success can be predicted prior to execution, monitored during execution, and evaluated after execution. This technique allows for the coarse evaluation of the probability of plan success that can be refined as time permits.

This algorithm works by simulating the system forward using the probabilistic plant model and then evaluating how the plant might have actually evolved given the simulated data. These steps are made tractable by sampling possible system evolutions and then sampling possible plant evolutions. As the number of samples grows, the probability computed by this technique converges to the actual probability that the plan will succeed.

The probabilistic physical plant is modeled using probabilistic transitions along with hard constraints. Thus, to make the sampling algorithms efficient, this thesis presented a solver for a class of constraint optimization problems that arise in the simulation and sampling problems of the plan monitoring capability. The solver can generate a user-specified k-best solutions to the problem or sample a random solution from the probability space. This solver is innovative in three ways. It is novel in its encoding of the problem using an sd-DNNF representation, it is a novel use of the sd-DNNF as a medium for sampling solutions, and it uses a novel algorithm for extracting the k-best solutions from the sd-DNNF.

Finally, this thesis has shown how this solver can be used to compute the belief state update equations for a PCCA model. These equations themselves represent a novel contribution of the semantics of PCCA models with respect to state estimation, and the first approach to allow for non-uniform observation probabilities.

Appendix A

Best-Solution Algorithm

This chapter introduces an algorithm to extract the best solution from a valued sd-DNNF. This is a summary of prior work by [18] and [2]. To extract the best solution, we need to find the best selection. Intuitively, since two selections differ based on the choices made at the **Or** nodes, we want to choose the best child for every **Or** node.

To find this best selection, we apply three rules:

1. For each leaf node l, the value of the leaf node is $\mathcal{L}_{\mathbf{P}}(l)$.

2. For each **And** node a, the value of the **And** node is the combination of the value of all of its children using $\times$.

3. For each **Or** node o, we choose the best child v of o using $\max$ and the value of o is the value of v.

The best selection is then the selection that includes the best child of each **Or** node visited from the root. We visit the best child of an **Or** node and all children of **And** nodes. Fig. A-2 shows an example of a best selection for Fig. A-1. In Fig. A-2, we highlight the best subgraphs for each node with a solid line. The subgraph that starts at the root node o1 is the best overall selection.

In following these three rules, if we cache at each node the value of the best choice, then the parents of the node can make use of this value to compute their own best value. This is dynamic programming, and is similar to solving tree-structured valued CSPs. The algorithm visit each edge

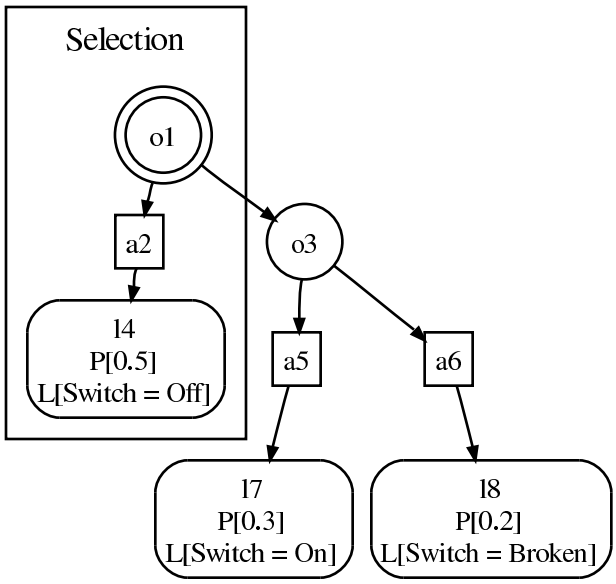

Figure A-1: This figure shows a selection of this simple sd-DNNF. The node o1 is the root of the tree. Our selection consists of o1, a2, and l4. This is a correct selection as it includes the root o1; it includes exactly one child of o1, namely a2; and it includes all of the children of a2, namely l4.

once: for **And** nodes we apply $\times$ to each child and for **Or** nodes we select the largest child with $\max$.

Since each parent needs their children's values to evaluate their own rule, we need a way of visiting all children before their parents. We have chosen to pre-order our nodes from 1 to $|V|$, such that the order of a node is greater than all parents of the node and less than all children of the node[1]. This is called a topological sort[15]. This ordering must exist because there are no cycles in the graph, though it is not in general unique. Using this ordering, the algorithm walks over the nodes from $|V|$ to 1 while applying the selection rules. This guarantees that every child is visited before its parent. We designate our ordered nodes V_O. This ordering always places the root r at position 1.

Once the algorithm has the best selection, it extracts the corresponding solution. For each **Or** node o in the graph, it records a decision $\eta\,(o) \in \text{Children}\,(o)$. The solution of the selection is the

[1]Recall that we are interested in solving the same problem multiple times, varying only the values, not the structure, so we can omit this sorting cost from our calculations.

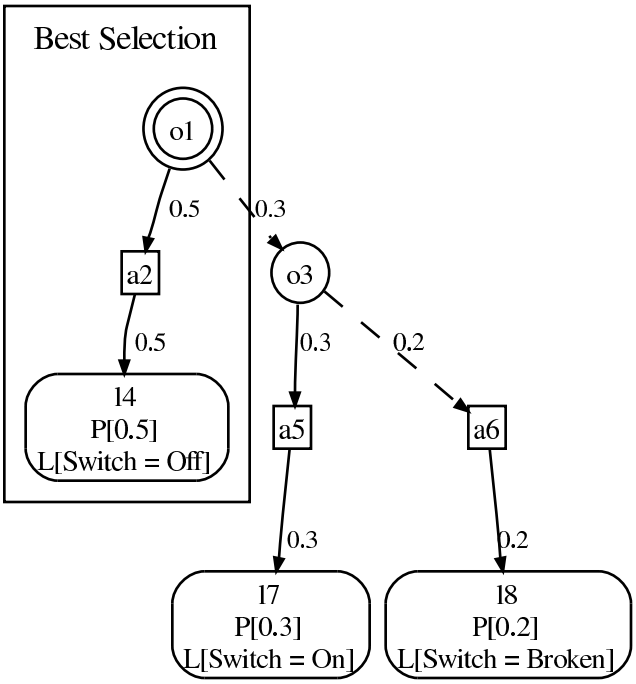

Figure A-2: The best selection for a simple sd-DNNF. Solid arcs represent the best choice for each node locally. Starting at the root, the best choice is o1, a2, and then l4, which is our best selection. We label the arcs with the value of the child.

set of $\mathcal{L}_L(l)$ for each l that has a path from the root r to l such that every **Or** node o_i in the path at position i is followed by $\eta(o_i)$ at position $i+1$.

Since we are looking for all leaves connected to the root by some path, this problem is naturally related to the transitive closure[16] of the sd-DNNF graph. A transitive closure of a directed graph is a new graph where the nodes are the same, but there is an edge $\langle m, n \rangle$ in the new graph if there is a *path* in the original graph from m to n. The problem of determining the leaves of the selection is equivalent to examining the leaves that are directly connected to the root node r in the transitive closure graph of a modified sd-DNNF, where the sd-DNNF is "modified" such that the only outgoing edge of an **Or** node is the one specified by η. We are only interested in the edges of the root node in the transitive closure graph, so we need not compute the full transitive closure of the graph. Since our graph is acyclic, we can use our topological ordering to walk once over the nodes and be

sure to visit all nodes along any path from the root to the leaves before their children. This lets our algorithm avoid adding edges explicitly, instead it only marks nodes that are connected to the root in the transitive closure graph.

A transitive closure includes the initial node, so we always mark the root. For connected **And** nodes, the algorithm marks all children as also connected to the root, as all of them are part of at least one path from the root to a leaf. For connected **Or** nodes, it marks only the child specified by η. Once the algorithm has marked the sd-DNNF, it applies $\mathcal{L}_L$ to all of the marked leaves and takes the union of these labels to produce the solution. Note that in Alg. A.3, $\mathcal{L}_L$ is applied to a leaf when it is visited, rather than making a second pass of the marked sd-DNNF.

A.1 Find-Best-Solution Algorithm

Algorithm A.1: FindBestSolution(V_O, E, $\mathcal{L}_L$, $\mathcal{L}_{\mathbf{P}}$, $\times$, $\max$)

```
1 η ← FindBestSelection(V_O, E, L_P, ×, max) ;
2 S ← GetSolutionFromSelection(V_O, E, L_L, η) ;
3 return S;
```

The algorithm that computes the best solution is shown in Alg. A.1. The algorithm is broken into the two passes specified above, a pass from the leaves to the root that computes the best selection and a second pass from the root to the leaves that extracts the solution of the best selection. The first pass returns $\eta : O \rightarrow V$, a function that records the best child node for each **Or** node. η defines a superset of a selection, as it contains decisions for **Or** nodes that are not part of the selection. The parts of η that are not part of the best selection are ignored by GetSolutionFromSelection as the irrelevant **Or** nodes are not connected to the root node. The second pass returns the best solution, a set of labels, corresponding to the selection.

A.1.1 Find-Best-Selection Algorithm

The first part of Alg. A.1 is shown in Alg. A.2. This function propagates the values of the leaves of the valued sd-DNNF to the root, making decisions at each **Or** node as to which child is best.

Algorithm A.2: FindBestSelection(V_O, E, $\mathcal{L}_{\mathbf{P}}$, $\times$, max)

```
1  for i = |V| to 1 do
2      v ← V_O [i] ;
3      switch v in
4          case v ∈ L                                  // Apply Rule 1 to leaf
5              P_V (v) ← L_P (v) ;
6          end
7          case v ∈ A                            // Apply Rule 2 to And node
               // Find best combination of the children of v
8              e ← some ⟨v, n⟩ ∈ E ;
9              p ← P_V (n) ;
10             foreach ⟨v, n⟩ ∈ E \ e do
11                 p ← p × P_V (n) ;
12             end
13             P_V (v) ← p ;
14         end
15         case v ∈ O                             // Apply Rule 3 to Or node
               // Find the best child of v
16             e ← some ⟨v, n⟩ ∈ E ;
17             ⟨b, p⟩ ← ⟨n, P_V (n)⟩ ;
18             foreach ⟨v, n⟩ ∈ E \ e do
19                 if P_V (n) max p then
20                     ⟨b, p⟩ ← ⟨n, P_V (n)⟩ ;
21                 end
22             end
23             P_V (v) ← p ;
24             η (v) ← b                                // Record best child
25         end
26     end
27 end
28 return η ;
```

This algorithm applies the three rules on page 185. Line 5 applies Rule 1. Lines 8-13 apply Rule 2. These lines combine the values of each child of the **And** node v using $\times$; the result is the value for v. Lines 16-23 apply Rule 3. These lines look for the best child of the **Or** node v, using $\max$. Line 24 then records the **Or** node's best child in η. Finally, the algorithm returns η on Line 28.

Runtime Analysis This algorithm visits every node once and every edge once. Nodes are stored sorted in an array, making $V_O[i]$ an $O(1)$ operation. It applies $\times$ or $\max$ per edge; these are both $O(1)$ operations. The algorithm stores edges with the parent node, and directly accesses the list of edges $v \rightarrow n$ on lines 10 and 18 in $O(1)$ time. $\mathbf{P}_V$ is stored with each node. As a space optimization, the algorithm uses $\mathcal{L}_{\mathbf{P}}(l)$ as $\mathbf{P}_V(l)$ for all the leaf nodes. Storing $\mathbf{P}_V$ with each node makes looking up and updating $\mathbf{P}_V$ also an $O(1)$ operation. Finally, we also store η with the **Or** nodes, likewise giving us $O(1)$ access. We can return η to the second part of Alg. A.1, Alg. A.3, by just passing Alg. A.3 our annotated sd-DNNF. Thus, for each edge and each node, we perform an $O(1)$ operation, giving Alg. A.2 a time complexity of $O(|E| + |V|)$. Since every node v in the sd-DNNF has a path from r to v, the sd-DNNF has at least as many edges as a tree. A tree has one more node than edge, so for the sd-DNNF $|E| + 1 \geq |V|$. This constraint lets us simplify our complexity bound to $O(|E|)$.

Space Analysis The sd-DNNF itself requires $O(|E| + |V|)$ space. The algorithm stores a value $\mathbf{P}_V$ per node and a reference to a node for η per **Or** node. This is an $O(|V|)$ additional space requirement.

A.1.2 Get-Solution-From-Selection Algorithm

The second part of Alg. A.1 is shown in Alg. A.3. This function extracts the solution that corresponds to the best selection η we found in Sect A.1.1. Line 1 initially marks the root node, in preparation for finding the leaves connected to the root in the modified sd-DNNF. Line 2 initially sets the solution to empty. Lines 3-20 then loop over the nodes from the root to the leaves. Line 5 ensures that the algorithm only extends paths from marked nodes, that is nodes that are already part of some path from the root. Line 8 adds to our solution by applying $\mathcal{L}_L$ to a marked leaf. Lines

Algorithm A.3: GetSolutionFromSelection(V_O, E, $\mathcal{L}_L$, η)

```
1  Marked ← {r} ;                    // Initially just the root is marked
2  S ← ∅ ;
3  for i = 1 to |V| do
4      v ← V_O [i] ;
5      if v ∈ Marked then
           // Extend path from marked node to selected children
6          switch v do
7              case v ∈ L          // Collect labels of selected leaf nodes
8                  S ← S ∪ {L_L (v)} ;
9              end
10             case v ∈ A                    // Mark all children of And node
11                 foreach ⟨v, n⟩ ∈ E do
12                     Marked ← Marked ∪ {n} ;
13                 end
14             end
15             case v ∈ O                   // Mark selected child of Or node
16                 Marked ← Marked ∪ {η (v)} ;
17             end
18         end
19     end
20 end
21 return S ;
```

11-13 marks all the children of a marked **And** node. Line 16 marks the one selected child of a marked **Or** node. Once lines 3-20 have visited all nodes, all of the marked leaves have been visited, hence S represents the solution corresponding to the selection; the algorithm returns this on Line 21.

Runtime Analysis This algorithm visits every node once and every edge of every marked node once. It stores a flag with each node indicating whether or not it is marked, thus setting and checking this flag is $O(1)$. Since it stores the flag per node, Line 1 is an $O(|V|)$ operation, as the algorithm must clear the marks on every node except the root, which must be set. We assume solution labels are unique and that the order in which they need to be returned is unimportant, hence adding $\mathcal{L}_L(v)$ to S on Line 8 just involves appending the symbol to a list, an $O(1)$ operation. The algorithm only marks those nodes that are part of the selection, hence this append operation is performed $O(|\text{Leaves in the Selection}|)$ times by this algorithm. As stated in Section A.1.1, the algorithm stores η with the **Or** nodes and edges with the parent node, thus all the operations performed per edge and per node are $O(1)$. Since the algorithm only marks nodes that are part of the selection, it only visits those edges that are part of the selection. Thus, the time complexity of this algorithm is $O(|\text{Edges in the Selection}| + |V|)$. Since the number of edges required to define a selection varies widely from sd-DNNF to sd-DNNF, we cannot further simplify this bound.

Note that an alternative formulation of this algorithm is a recursive depth-first walk from the root to the leaves, visiting all the marked nodes. Due to the decomposition and determinism of the sd-DNNF, a node that is part of a selection always has exactly one parent that is part of a selection (except the root, which has none). Thus, the algorithm does not visit the same node more than once. This formulation only visits those nodes that are part of the selection, reducing the complexity of the algorithm to $O(|\text{Edges in the Selection}| + |\text{Nodes in the Selection}|)$. Since this forms a tree, $|\text{Edges in the Selection}| + 1 = |\text{Nodes in the Selection}|$, hence this simplifies to $O(|\text{Nodes in the Selection}|)$. While this is likely a superior implementation, the GetSolutionFromSelection Algorithm contributes insignificantly to the time and space complexity of the FindBestSolution Algorithm. The FindBestSolution Algorithm is dominated by the FindBestSelection algorithm, hence we have chosen not to investigate this improved algorithm further for the purpose of this

thesis.

Space Analysis The sd-DNNF itself requires $O(|E| + |V|)$ space. The algorithm stores a flag per node for Marked. It also stores a list of symbols (or references to symbols) in S, the solution. The flags require $O(|V|)$ space and S requires $O(|\text{Leaves in the Selection}|)$ space.

The alternative formulation requires a stack for the **And** nodes along the current path, recording which child is currently being visited. This stack contains at most the number of **And** nodes along the path with the most **And** nodes. This is clearly no more than $|A|$ as opposed to storing $|V|$ flags.

Putting together the runtime and space analysis for Algorithms A.2 and A.3, we now state the requirements for Algorithm A.1. The time required is dominated by Alg. A.2, requiring $O(|E| + |V|)$ time, and thus this is also the time required by Alg. A.1. The space required is proportional to the number of nodes in the graph, plus the graph itself, so $O(|E| + |V|)$ total space is used.

A.2 Find-Best-Solution Example

We now show, as an example, the two parts of Alg. A.1 running on the example shown in Fig. A-2. The progression of Alg. A.2 is show in Figures A-3, A-4, and A-5. Figures A-6, A-7, and A-8 show progressively how Alg. A.3 operates on Fig. A-2.

The example shown in Fig. A-2 is defined by the following sd-DNNF:

- The nodes $V = \{\text{o1}, \text{a2}, \text{o3}, \text{l4}, \text{a5}, \text{a6}, \text{l7}, \text{l8}\}$, where $A = \{\text{a2}, \text{a5}, \text{a6}\}$, $O = \{\text{o1}, \text{o3}\}$, and $L = \{\text{l4}, \text{l7}, \text{l8}\}$. The root node $r = \text{o1}$. The number at the end of each node's name in V is its ordering by V_O.
- The edges E are $\langle \text{o1}, \text{a2} \rangle$, $\langle \text{o1}, \text{o3} \rangle$, $\langle \text{a2}, \text{l4} \rangle$, $\langle \text{o3}, \text{a5} \rangle$, $\langle \text{o3}, \text{a6} \rangle$, $\langle \text{a5}, \text{l7} \rangle$, and $\langle \text{a6}, \text{l8} \rangle$.
- The symbols $\mathcal{L}_L$ are:
  - $\mathcal{L}_L\,(\text{l4}) =$ "Switch = Off"
  - $\mathcal{L}_L\,(\text{l7}) =$ "Switch = On"
  - $\mathcal{L}_L\,(\text{l8}) =$ "Switch = Broken"
- The values $\mathcal{L}_{\mathbf{P}}$ are: $\mathcal{L}_{\mathbf{P}}\,(\text{l4}) = 0.5$, $\mathcal{L}_{\mathbf{P}}\,(\text{l7}) = 0.3$, and $\mathcal{L}_{\mathbf{P}}\,(\text{l8}) = 0.2$.
- Arithmetic multiplication for $\times$.
- Arithmetic greater-than for $\max$.

A.2.1 Find-Best-Selection Example

Recall that the FindBestSelection algorithm employs dynamic programming to ensure that the value of each node is only computed once. It stores the computed value in the variable $\mathbf{P}_V$. The algorithm decides the best selection locally at each **Or** node, o1 and o3 in this example, based on the values of its children. This selection is stored in the variable η (Eta). The initially empty state of these variables and the graph are shown in Fig. A-3.

Alg. A.2 consists of one loop that runs from the leaves to the root of the valued sd-DNNF. The first node assigned to v on Line 2 is l8. This is a leaf node, so the algorithm executes Line 5.

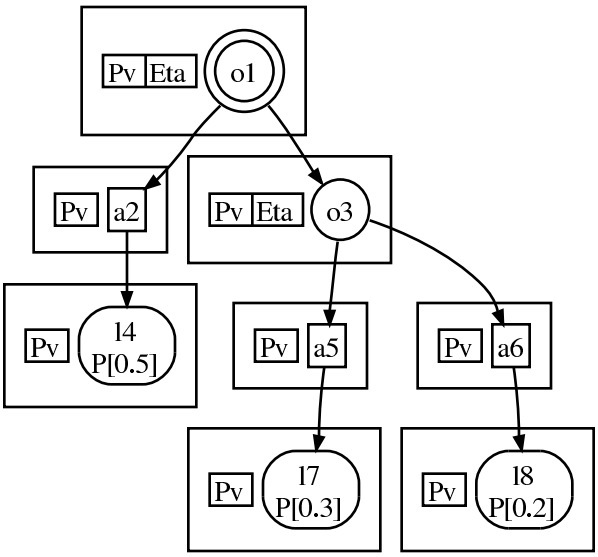

Figure A-3: This figure shows the initial state of the FindBestSelection function for this simple sd-DNNF. The values of $\mathbf{P}_V$ are initially unknown and η is initially undecided.

This sets $\mathbf{P}_V\,(\text{l8}) = \mathcal{L}_{\mathbf{P}}\,(\text{l8}) = 0.2$. The next v is l7, which sets $\mathbf{P}_V\,(\text{l7}) = 0.3$. The algorithm then visits $v = \text{a6}$, which executes lines 8 to 13. The only edge of the form $\langle \text{a6}, *\rangle$, i.e. the only out-going edge of a6, is the edge $\langle \text{a6}, \text{l8}\rangle$. Since $\mathbf{P}_V\,(\text{l8}) = 0.2$, the algorithm sets $p = 0.2$ and then sets $\mathbf{P}_V\,(\text{a6}) = 0.2$. It continues, by visiting $v = \text{a5}$, setting $\mathbf{P}_V\,(\text{a5}) = 0.3$, visiting $v = \text{l4}$ and finally setting $\mathbf{P}_V\,(\text{l4}) = 0.5$. Fig. A-4 shows the state of $\mathbf{P}_V$ and η at this point.

The algorithm then visits o3. The node o3 is our first **Or** node, and visiting this node executes lines 16 to 24. The node o3 has two children, a5 and a6. Lets assume that $n = \text{a5}$ is first, so the algorithm sets $b = \text{a5}$ and $p = \mathbf{P}_V\,(\text{a5}) = 0.3$ on Line 17. It then visits $n = \text{a6}$ and skips this node, because $\mathbf{P}_V\,(\text{a6}) = 0.2$ is less than 0.3. Line 23 then sets $\mathbf{P}_V\,(\text{o3}) = 0.3$, the value of o3's best child. Finally, Line 24 sets $\eta\,(\text{o3}) = \text{a5}$, recording the best choice. The algorithm then continues on to the last two nodes, a2 and o1. It visits the node a2 and sets $\mathbf{P}_V\,(\text{a2}) = 0.5$, and then visits the node o1 and sets $\mathbf{P}_V\,(\text{o1}) = 0.5$ and $\eta\,(\text{o1}) = \text{a2}$. This is the final state of the algorithm, as shown in Fig. A-5. The algorithm now returns η on Line 28, where $\eta\,(\text{o1}) = \text{a2}$ and $\eta\,(\text{o3}) = \text{a5}$.

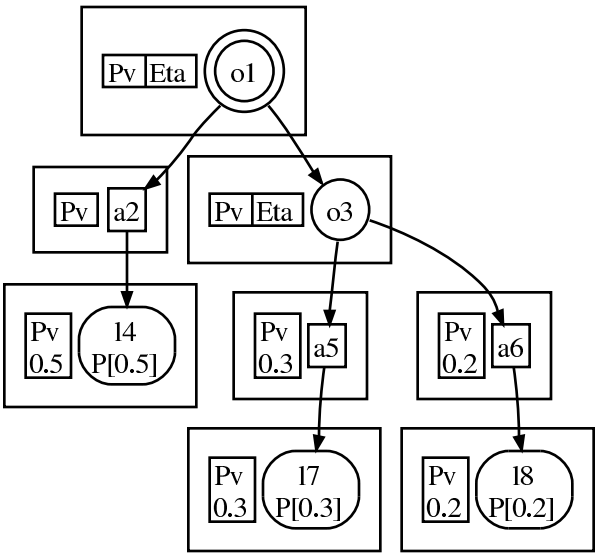

Figure A-4: This figure shows the intermediate state of the FindBestSelection function for this simple sd-DNNF. We propagated the leaves to the **And** nodes using lines 8-13 of Alg. A.2.

A.2.2 Get-Solution-From-Selection Example

The GetSolutionFromSelection algorithm, Alg. A.3, determines the leaves connected to the root in the valued sd-DNNF modified by η. The algorithm marks all the nodes that have a path from the root to themselves, and it records which nodes are marked in the Marked variable. The algorithm stores the set of symbols of the solution in the variable S. Initially, the root o1 is marked, hence Marked= $\{\text{o1}\}$. The initial state at the start of the main loop on Line 3 is shown in Fig. A-6. We denote membership in Marked by coloring the marked nodes black.

The main loop runs from the root down to the leaves, hence the first node visited is the root o1. The node o1 is marked, as we stated initially, and is an **Or** node. We thus execute Line 16. Since $\eta\,(\text{o1}) = \text{a2}$, we add a2 to Marked. Marked is now $\{\text{o1, a2}\}$. This state is shown in Fig. A-7.

The algorithm then visits the node a2, which is marked, and execute the lines 11 to 13. This marks all of the children of a2, in this case only l4. Thus, after executing lines 11 to 13, Marked is now $\{\text{o1, a2, l4}\}$. It then visits o3, but o3 is not marked, so the algorithm skips o3. The node l4 is

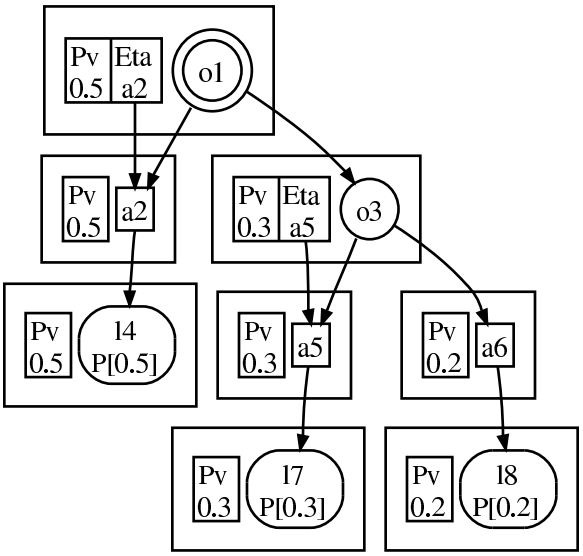

Figure A-5: This figure shows the final state of the FindBestSelection function for this simple sd-DNNF, just prior to returning η. The algorithm propagated the values of $\mathbf{P}_V$ to the root using lines 16-23 of Alg. A.2. It also set η for both **Or** nodes using Line 24 of Alg. A.2.

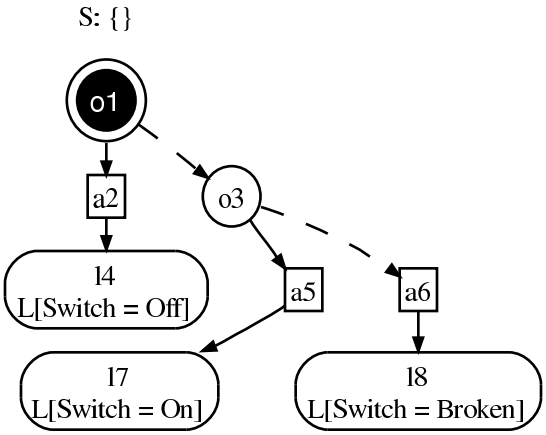

Figure A-6: This figure shows the initial state of the GetSolutionFromSelection function for this simple sd-DNNF, just after executing lines 1 and 2 of Alg. A.3. Initially the only marked node is the root o1. Marked nodes are black, while the remaining white nodes are not marked. The solid lines connecting the nodes represent the edges that are part of the modified sd-DNNF, while the dashed lines are currently suppressed by the **Or** node choice stored in η.

then visited, executing Line 8. Since $\mathcal{L}_L$ (l4) = "Switch = Off", the algorithm adds this symbol to S: $S = \{$"Switch = Off"$\}$. The nodes a5, a6, l7, and l8 are then visited in that order, but none of them are marked. The main loop is now finished and the algorithm is ready to return S on Line 21. This state is shown in Fig. A-8.

A.3 Summary

This chapter described the prior work of [18] and [2], an algorithm for extracting the best solution from a valued sd-DNNF. The algorithm requires $O(|E|+|V|)$ time and space. The algorithm works in two parts, the first part passes from the leaves to the root, deciding along the way which sub-tree of **Or** nodes is the optimal choice while propagating the value of the sub-trees to the root. The second part uses the selection of the first part, which is defined by η, to extract a solution.

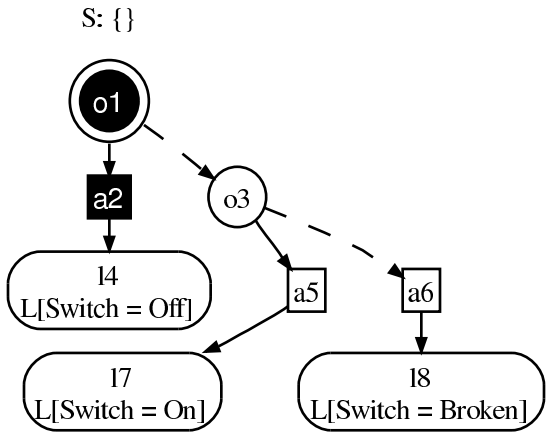

Figure A-7: This figure shows the state of the GetSolutionFromSelection function for this simple sd-DNNF after executing Line 16 of Alg. A.3 with $v = \text{o1}$ on Fig. A-6. This marks a2 as $\eta\,(\text{o1}) =$ a2. Marked nodes are black, while the remaining white nodes are not marked. The solid lines connecting the nodes represent the edges that are part of the modified sd-DNNF, while the dashed lines are currently suppressed by the **Or** node choice stored in η.

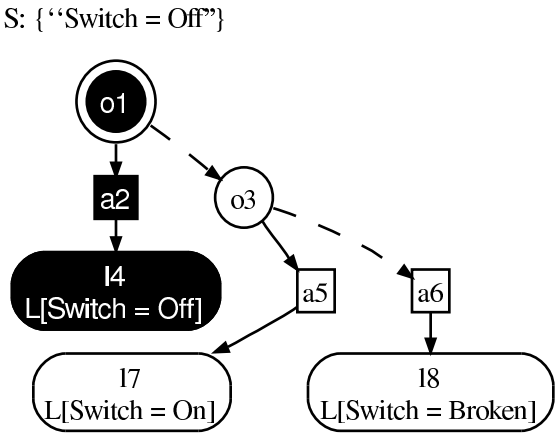

Figure A-8: This figure shows the final state of the GetSolutionFromSelection function for this simple sd-DNNF. After Fig. A-7, the algorithm has executed Line 12 of Alg. A.3 with $v =$ a2 and $n =$ l4, thus marking l4. It then executed Line 8 with $v =$ l4, adding “Switch = Off” to our solution S. This S is then returned on Line 21. Marked nodes are black, while the remaining white nodes are not marked. The solid lines connecting the nodes represent the edges that are part of the modified sd-DNNF, while the dashed lines are currently suppressed by the **Or** node choice stored in η.

Bibliography

[1] *Int. Symp. on Artificial Intelligence, Robotics and Automation in Space*, St-Hubert, Canada, June 2001.

[2] Anthony Barrett. Model compilation for real-time planning and diagnosis with feedback. In Kaelbling and Saffiotti [34], pages 1195–1200.

[3] L. Baum and T. Petrie. Statistical inference for probabilistic functions of finite-state Markov chains. *Annals of Mathematical Statistics*, 37:1554–1563, 1966.

[4] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, London, UK, 1999. Springer-Verlag.

[5] S. Bistarelli, U. Montanari, F. Rossi, T. Schiex, G. Verfaillie, and H. Fargier. Semiring-based csps and valued csps: Frameworks, properties,and comparison. *Constraints*, 4(3):199–240, 1999.

[6] Steve Block, Andreas F. Wehowsky, and Brian C. Williams. Robust execution of contingent temporally flexible plans. In *AAAI*, pages 802–808. AAAI Press, 2006.

[7] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[8] Seung Chung. A Decomposed Symbolic Approach to Reactive Planning. Master's thesis, Massachusetts Institute of Technology, MIT Space Engineering Research Center, June 2003. SSL #7-03.

[9] A. Cimatti, E. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Integrating BDD-based and SAT-based Symbolic Model Checking. In *Proc. "Frontiers of Combining Systems, FROCOS'02"*, volume 2309 of *LNAI*, Santa Margherita, Italy, January 2002. Springer.

[10] A. Cimatti, M. Pistore, M. Roveri, and R. Sebastiani. Improving the Encoding of LTL Model Checking into SAT. In *Proc. workshop on Verification Model Checking and Abstract Interpretation, VMCAI'02*, volume 2294 of *LNCS*, Venice, Italy, January 2002. Springer.

[11] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[12] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[13] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*, pages 1060,1066. In [17], 2000. Harmonic Series.

[14] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*, pages 29,142. In [17], 2000. Merge Sort.

[15] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*, pages 549–551. In [17], 2000. Topological Sort.

[16] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*, pages 632–633. In [17], 2000. Transitive Closure.

[17] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 2000.

[18] Adnan Darwiche. Decomposable negation normal form. *J. ACM*, 48(4):608–647, 2001.

[19] Adnan Darwiche. C2D v2.20. http://reasoning.cs.ucla.edu/c2d, July 2005.

[20] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *J. Artif. Intell. Res. (JAIR)*, 17:229–264, 2002.

[21] Leonardo de Moura, Sam Owre, and N. Shankar. The sal language manual. Csl technical report, SRI, August 2003.

[22] Thomas Linus Dean. *Temporal imagery: an approach to reasoning about time for planning and problem solving*. PhD thesis, New Haven, CT, USA, 1986.

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

[24] Robert Effinger. Optimal temporal planning at reactive time scales via dynamic backtracking branch and bound. Master's thesis, Massachusetts Institute of Technology, September 2006.

[25] Paul Elliott. Extracting the K Best Solutions from a Valued And-Or Acyclic Graph. Master's thesis, Massachusetts Institute of Technology, CSAIL, May 2007.

[26] E.M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. In David L. Dill, editor, *Proceedings of the sixth International Conference on Computer-Aided Verification CAV*, volume 818, pages 415–427, Standford, California, USA, 1994. Springer-Verlag.

[27] R. Gerth, Den Dolech Eindhoven, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *In Protocol Specification Testing and Verification*, pages 3–18. Chapman & Hall, 1995.

[28] Dimitra Giannakopoulou and Klaus Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In *Proc. ASE'01: 16th IEEE Int'l Conf. on Automated Software Engineering*, pages 412–423, Coronado Island, CA, November 2001.

[29] Klaus Havelund and Grigore Rosu. Testing linear temporal logic formulae on finite execution traces. Technical report, May 2001.

[30] Andreas Hofmann. *Robust Execution of Bipedal Walking Tasks from Biomechanical Principles*. PhD thesis, Massachusetts Institute of Technology, January 2006.

[31] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[32] Michel Ingham, Robert Ragno, and Brian Williams. A Reactive Model-based Programming Language for Robotic Space Explorers. In *ISAIRAS-01* [1].

[33] J.R. Burch, E.M. Clarke, D.E. Long, K.L. MacMillan, and D.L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, 1994.

[34] Leslie Pack Kaelbling and Alessandro Saffiotti, editors. *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30-August 5, 2005*. Professional Book Center, 2005.

[35] Phil Kim, Brian C. Williams, and Mark Abramson. Executing reactive, model-based programs through graph-based temporal planning. In Nebel [46], pages 487–493.

[36] M. Kwiatkowska, G. Norman, and D. Parker. Verifying randomized distributed algorithms with PRISM. In *Proc. Workshop on Advances in Verification (Wave'2000)*, July 2000.

[37] M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic symbolic model checker. In P. Kemper, editor, *Proc. Tools Session of Aachen 2001 International Multiconference on Measurement, Modelling and Evaluation of Computer-Communication Systems*, pages 7–12, September 2001. Available as Technical Report 760/2001, University of Dortmund.

[38] M. Kwiatkowska, G. Norman, and D. Parker. Game-based abstraction for Markov decision processes. In *Proc. 3rd International Conference on Quantitative Evaluation of Systems (QEST'06)*, pages 157–166. IEEE CS Press, 2006.

[39] M. Kwiatkowska, G. Norman, and D. Parker. Stochastic model checking. In M. Bernardo and J. Hillston, editors, *Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM'07)*, volume 4486 of *LNCS (Tutorial Volume)*, pages 220–270. Springer, 2007.

[40] J. Larrosa and R. Dechter. The dual representation of non-binary semiring-based csps, 2000.

[41] Oliver Martin. Accurate belief state update for probabilistic constraint automata. Master's thesis, Massachusetts Institute of Technology, MIT MERS, June 2005.

[42] Oliver Martin, Michel Ingham, and Brian Williams. Diagnosis as Approximate Belief State Enumeration for Probabilistic Concurrent Constraint Automata. In *Proceedings of the AAAI*, 2005.

[43] Kenneth L. McMillan. *Symbolic Model Checking*. PhD thesis, Carnegie Mellon University, Department of Computer Science, 1992.

[44] Kenneth L. McMillan. Interpolation and sat-based model checking. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003.

[45] Paul H. Morris, Nicola Muscettola, and Thierry Vidal. Dynamic Control of Plans with Temporal Uncertainty. In Nebel [46], pages 494–502.

[46] Bernhard Nebel, editor. *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001*. Morgan Kaufmann, 2001.

[47] H. E. Rauch, G. Tung, and C. T. Striebel. Maximum likelihood estimates of linear dynamic systems. *American Institute of Aeronautics and Astronautics Journal (AIAAJ)*, 3(8):1445–1450, 1965.

[48] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*, pages 511–519. Prentice Hall, Upper Saddle River, NJ, 2nd edition edition, 2003.

[49] Martin Sachenbacher and Brian C. Williams. On-demand bound computation for best-first constraint optimization, 2004.

[50] Martin Sachenbacher and Brian C. Williams. Solving soft constraints by separating optimization and satisfiability, October 2005.

[51] Thomas Schiex, Hélène Fargier, and Gerard Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. In Chris Mellish, editor, *IJCAI'95: Proceedings International Joint Conference on Artificial Intelligence*, Montreal, 1995.

[52] John Stedl. Managing temporal uncertainty under limited communication: A formal model of tight and loose team communication. Master's thesis, Massachusetts Institute of Technology, September 2004.

[53] Moshe Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *LICS*, pages 332–344, 1986.

[54] Thierry Vidal and Hélène Fargier. Contingent Durations in Temporal CSPs: From Consistency to Controllabilities. In *TIME*, pages 78–85. IEEE Computer Society, 1997.

[55] Thierry Vidal and Malik Ghallab. Dealing with Uncertain Durations in Temporal Constraint Networks Dedicated to Planning. In Wolfgang Wahlster, editor, *ECAI*, pages 48–54. John Wiley and Sons, Chichester, 1996.

[56] Brian C. Williams, Michel Ingham, Seung H. Chung, and Paul H. Elliott. Model-based Programming of Intelligent Embedded Systems and Robotic Space Explorers. In *Proceedings of the IEEE*, volume 9, pages 212–237, Jan 2003.

[57] Brian C. Williams and Michel D. Ingham. Model-based Programming: Controlling Embedded Systems by Reasoning About Hidden State. In *Eighth Int. Conf. on Principles and Practice of Constraint Programming*, Ithaca, NY, September 2002.

[58] Brian C. Williams, Phil Kim, Michael Hofbaur, Jon How, Jon Kennell, Jason Loy, Robert Ragno, John Stedl, and Aisha Walcott. Model-based reactive programming of cooperative vehicles for mars exploration. In *ISAIRAS-01* [1].

[59] Brian C. Williams and Robert J. Ragno. Conflict-directed a* and its role in model-based embedded systems. *Discrete Appl. Math.*, 155(12):1562–1595, 2007.