BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

OPTIMIZING INDUCTIVE CONTROLLER SYNTHESIS METHODS FOR POMDPS WITH DISCOUNTED REWARDS PROPERTIES

OPTIMALIZACE INDUKTIVNÍ SYNTÉZY KONTROLÉRŮ PRO POMDP S ČASOVĚ OMEZENÝMI CENAMI

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR
AUTOR PRÁCE

ONDŘEJ KŘÍŽ

SUPERVISOR
VEDOUCÍ PRÁCE

doc. RNDr. MILAN ČEŠKA, Ph.D.

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

148661

Institut:	Department of Intelligent Systems (UITS)
Student:	**Kříž Ondřej**
Programme:	Information Technology
Specialization:	Information Technology
Title:	**Optimizing Inductive Controller Synthesis Methods for POMDPs with Discounted Rewards Properties**
Category:	Formal Verification
Academic year:	2022/23

Assignment:

1. Study state-of-the-art controller synthesis methods for Partially Observable MDPs (POMDPs) with the focus on inductive synthesis and discounting reward specifications
2. Design optimisations improving inductive synthesis methods for discounting reward specifications.
3. Implement the optimisations within the tool PAYNT.
4. Using suitable benchmarks, perform a detailed experimental evaluation of the implemented optimisations.

Literature:

- Kochenderfer, M.J., Wheeler, T.A., and Wray K.H, Algorithms for Decision Making, MIT Press 2021.
- Andriushchenko, R., Češka, M., Junges, S., and Katoen, J.P. Inductive synthesis of finite-state controllers for POMDPs. In UAI'22. Proceedings of Machine Learning Research.
- Andriushchenko, R., Češka, M., Junges, S., Katoen, J.P. and Stupinský, Š. PAYNT: A Tool for Inductive Synthesis of Probabilistic Programs. In *CAV 2021*. Springer.
- Hensel, C., Junges, S., Katoen, J.-P., Quatmann, T., and Volk, M. The probabilistic model checker Storm. *Int. J. Softw. Tools Technol. Transf.*2022.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Češka Milan, doc. RNDr., Ph.D.**
Consultant:	Ing. Roman Andriushchenko
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	10.5.2023
Approval date:	3.11.2022

Abstract

Probabilistic model checking is essential for verifying systems in diverse domains. A key limitation of the PAYNT tool, which synthesises probabilistic programs satisfying given specifications, lies in its handling of discounted properties. This thesis extends the STORM framework upon which PAYNT is built, incorporating the discounted value iteration method within inductive synthesis process to address this issue. The discounted value iteration function was developed within STORM, involving solver environment identification, decision-making segments in the code, and Gauss-Seidel multiplication for enhanced computational capabilities. The necessity for a PRISM formula in PAYNT's model checking process presented challenges for bypassing the discount factor transformation step. To overcome this, a discount factor transformation with a factor close to one was employed, comparing potential optima vectors between discounted and undiscounted iterations. This study improves discounted property handling in PAYNT and the STORM framework, providing a foundation for further advancements in the development of PAYNT.

Abstrakt

Pravděpodobnostní kontrola modelů je nezbytnou součástí verifikace systémů v různých prostředích. Klíčová limitace nástroje PAYNT, který syntetizuje pravděpodobnostní programy splňující danou specifikaci, leží v jeho zacházení s diskontními vlastnostmi. Tato práce rozšiřuje rámec nástroje STORM, na němž je postaven PAYNT, implementací diskontní iterace hodnot v rámci procesu induktivní syntézy. Diskontní iterace hodnot byla implementována v rámci STORMu, včetně identifikace vhodného prostředí pro řešitele, rozhodovacích segmentů v kódu a Gauss-Seidelovým násobením pro vylepšené výpočetní schopnosti. Nezbytnost použití vzorce v jazyce PRISM v rámci kontroly modelů v PAYNTu představuje problém pro vynechání diskontní transformace, která ztěžuje kontrolu modelu. Proto byla diskontní transformace ponechána s diskontním faktorem blízkým k jedné, a jsou porovnávány hodnoty potenciálních optim, které vrací diskontní a nediskontní iterace hodnot. Tato práce zlepšuje práci PAYNTu a STORMu s diskontními hodnotami a poskytuje základ pro další pokroky ve vývoji PAYNTu a STORMu.

Keywords

Partially observable Markov decision processes, inductive synthesis, discounted reward properties

Klíčová slova

Markovské rozhodovací procesy s částečným pozorováním, induktivní syntéza, časově omezené ceny

Reference

KŘÍŽ, Ondřej. *Optimizing Inductive Controller Synthesis Methods for POMDPs with Discounted Rewards Properties.* Brno, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. RNDr. Milan Češka, Ph.D.

Rozšířený abstrakt

Ve světě plném nejistoty stále narůstá potřeba přesného modelování systémů s pravděpodobnostní nejistotou. Pravděpodobnostní ověřování modelů je technika, která se nachází na pomezí umělé inteligence, informatiky a formálních metod. Je převážně používáno k ověřování správnosti systémů, které zahrnují pravděpodobnostní nejistotu, a nachází uplatnění v různých oblastech, jako jsou počítačové sítě, kybernetická bezpečnost, robotika, vestavěné systémy a dokonce i biologické modelování. Vzhledem k rostoucí složitosti systémů poskytuje přístup pravděpodobnostního ověřování modelů pevnou matematickou metodologii, která zajišťuje, že tyto systémy se chovají tak, jak se očekává za různých podmínek, a umožňuje odhalování potenciálních problémů před tím, než vedou k vážným selháním.

Implementace nástrojů pro pravděpodobnostní ověřování modelů umožňuje syntézu pravděpodobnostních programů, které splňují dané specifikace, čímž rozšiřuje použitelnost tohoto přístupu. Navíc, v době, kdy se stáváme stále více závislými na systémech umělé inteligence a strojového učení, nabývá pravděpodobnostní modelování ještě většího významu. Díky své schopnosti poskytovat podrobné informace o chování systémů přispívá tato ověřovací technika, spolu s pokroky v oblasti vysvětlitelné umělé inteligence, k vytvoření transparentních, srozumitelných a důvěryhodných systémů umělé inteligence.

Částečně pozorovatelné Markovovy rozhodovací procesy (POMDP) poskytují matematický rámec pro modelování rozhodovacích problémů, kde je stav systému nejistý. Hledání optimálních strategií pro tyto modely je klíčovým aspektem procesu pravděpodobnostního ověřování modelů. Je však důležité poznamenat, že nalezení optimální strategie pro POMDP je obecně nerozhodnutelné, což znamená, že neexistuje univerzální algoritmus, který by mohl určit optimální strategii pro každý možný POMDP [2]. To představuje významnou výpočetní výzvu a zdůrazňuje potřebu pokročilých technik v této oblasti.

Protipříklady řízená induktivní syntéza (CEGIS), která se používá k výpočtu optimálních strategií, zahrnuje uhádnutí kandidátního řešení, jeho ověření proti počáteční specifikaci a učení se z nesprávných odhadů pro zlepšení odhadů následujících. Hlavní výhodou tohoto přístupu je vyhnout se vyčerpávajícího procházení všemi možnými řešeními. PAYNT (Probabilistic progrAm sYNThesizer) je nástroj, který využívá řízenou induktivní syntézu [4].

V nástroji PAYNT funguje řízená induktivní syntéza jako metoda učení z analýzy potenciálních řešení nebo "realizací". "Učící se algoritmus" vybere jednu realizaci a předá ji "učiteli", který posoudí, zda daná realizace splňuje danou specifikaci. Pokud ne, učitel poskytne další informace, často ve formě protipříkladu [4].

Nicméně, metoda iterace hodnot v PAYNTu, která je základním stavebním prvkem při hledání optimálních strategií, není tak účinná při diskontních vlastnostech ve srovnání s dříve navrženým algoritmem SARSOP od Kurniawati a kolektivu, který pracuje s aproximací prostoru všech dosažitelných přesvědčení pod optimálními strategiemi [13].

V této práci představuji optimalizaci pro metody induktivní syntézy kontrolerů aplikované na částečně pozorovatelné Markovovy rozhodovací procesy (POMDP) s diskontními vlastnostmi. Můj přístup se zaměřuje na integraci nové metody do nástroje STORM, pravděpodobnostního ověřovače modelů, na kterém je PAYNT postaven. Tato metoda zlepšuje výkon iterace hodnot s diskontními vlastnostmi pomocí rychlého a efektivního násobení vektorů. Pro PAYNT je výsledkem optimalizovaný výpočet, který potenciálně obejde náročný výpočetní krok transformace kontrolovaného modelu do formy bez diskontního faktoru. Tato optimalizace má za cíl snížit čas ověřování modelu a zvýšit efektivitu hledání optimálních strategií.

Funkce iterace hodnot s diskontními vlastnostmi byla vyvinuta v rámci STORMu a zahrnuje identifikaci řešitelského prostředí, segmenty rozhodování v kódu a Gauss-Seidelovu metodu pro vylepšení výpočetních schopností. Nutnost použití PRISM formule v procesu modelového ověřování PAYNTu představovala výzvu při obejití kroku transformace modelu na bezdiskontní model. Pro překonání tohoto problému byla použita transformace slevového faktoru s faktorem blízkým jedné. Pro kontrolu správnosti postupu je prováděno porovnávání potenciálních optimálních vektorů mezi diskontní a bezdiskontní iterací hodnotami. Tato studie zlepšuje zpracování slevových vlastností v nástroji PAYNT, mírně zrychluje proces ověřování modelu a poskytuje základ pro další rozvoj nástroje PAYNT.

Optimizing Inductive Controller Synthesis Methods for POMDPs with Discounted Rewards Properties

Declaration

Acknowledgements

I would like to express my gratitude to my supervisor, RNDr. Milan Češka, Ph.D., for his guidance, support and advice throughout my work. I would also like to thank Ing. Roman Andriushchenko for his willingness and advice regarding technical part of my work. And finally I would like to thank my girlfriend, my friends and family for continuous support throughout this testing times.

Contents

List of Figures

Chapter 1

Introduction

In a world rife with uncertainty, the necessity for accurately modeling systems with probabilistic uncertainty is continually increasing. Probabilistic model checking is a technique situated at the intersection of artificial intelligence, computer science, and formal methods. It is primarily used for verifying the correctness of systems that include probabilistic uncertainty, and has found applications in diverse fields such as computer networks, cybersecurity, robotics, embedded systems, and even biological modeling. As systems grow increasingly complex, the probabilistic model checking approach offers a rigorous, mathematically grounded methodology to ensure that these systems behave as expected under a range of conditions, and to pinpoint potential issues before they lead to serious failures.

The implementation of probabilistic model checking tools allows for the synthesis of probabilistic programs that satisfy given specifications, thereby expanding the applicability of this approach. Furthermore, as we step into an era of increasing dependence on artificial intelligence and machine learning systems, the importance of probabilistic model checking, with its capacity to provide detailed insights into system behavior, becomes even more critical. This verification technique, in concert with advancements in explainable AI, contributes to the broader goal of creating transparent, understandable, and trustworthy AI systems.

Partially Observable Markov Decision Processes (POMDPs) provide a mathematical framework for modeling decision-making problems where the system state is uncertain. Finding optimal policies for these models is a critical aspect of the probabilistic model checking process. However, it is important to note that finding an optimal policy for a POMDP is generally undecidable, meaning there is no universal algorithm that can determine the optimal policy for every possible POMDP [2]. This presents a significant computational challenge and underscores the need for advanced techniques in the field.

Counter-example guided inductive synthesis (CEGIS), used for computing optimal policies, involves guessing a candidate solution, validating it against the initial specification, and learning from incorrect guesses to improve subsequent ones. This approach's primary advantage is avoiding exhaustive traversal of all possible solutions. PAYNT (Probabilistic progrAm sYNThesizer) is a tool that utilizes oracle-guided inductive synthesis [4].

In PAYNT, oracle-guided inductive synthesis operates as a method of learning from the analysis of potential solutions or „realizations“. A „learner“ selects a realization and passes it to an „oracle“ which evaluates whether the realization satisfies a given specification. If it doesn't, the oracle provides additional information, often in the form of a counter-example [4].

However, PAYNT's value iteration, a basic building block in finding optimal policies, falls short for discounted properties when compared to the earlier proposed SARSOP algorithm by Kurniawati et al., which works by approximating the space of all reachable belief states under optimal policies [13].

In this thesis, I present an optimization for inductive controller synthesis methods applied to Partially Observable Markov Decision Processes (POMDPs) with discounted rewards properties. My approach focuses on integrating a novel method within STORM, the probabilistic model checker that underpins PAYNT. This method enhances the performance of discounted value iteration by leveraging fast and effective vector multiplication. As a result, it offers PAYNT an optimized path that potentially bypasses the computationally demanding step of transforming the checked model into an undiscounted form. Consequently, this optimization aims to reduce the model checking time, increasing the efficiency of finding optimal policies.

Structure

This thesis is organized as follows: In Chapter 2, we lay the groundwork for the subsequent discussions and analyses by introducing the fundamental concepts and definitions relevant to our study, particularly Markov Decision Processes (MDPs) and partially observable MDPs. In Chapter 3, we delve into the complexities of Partially Observable Markov Decision Processes (POMDPs) and present Finite State Controllers and Inductive Synthesis as promising approaches for synthesizing policies for POMDPs, including an introduction to the concept of Abstraction Refinement. Then, in Chapter 4, having established the theoretical groundwork, we explore the practical aspects of model checking, focusing on tools significant for my work. In Chapter 5, we present the author's contributions to the development of the PAYNT tool, along with an experimental evaluation of the implemented enhancements, discussing their impact on the tool's capabilities and performance. In the final Chapter 6, we summarize the findings of the thesis, discuss potential future work in the area, and provide concluding remarks on the significance and implications of the research undertaken.

Chapter 2

Preliminaries

Following chapter lays the groundwork for the subsequent discussions and analyses by introducing the fundamental concepts and definitions relevant to my study. I'll start by providing an overview of the basic definitions and concepts, which form the foundation of decision-making problems in the context of Markov Decision Processes (MDPs). Then we will delve into problematics of MDPs and partially observable MDPs and inherent concepts important to my work. The theoretical concepts discussed in this chapter have been adapted from the book „Algorithms for Decision Making“ authored by M. J. Kochenderfer [12].

2.1 Markov Decision Processes

In this section, we introduce the foundational concepts of Markov Decision Processes (MDPs) and their underlying Markov Chains, covering states, actions, transitions, rewards, and policies, which together form the basis for understanding decision-making problems under uncertainty. These core ideas pave the way for our exploration of Partially Observable Markov Decision Processes (POMDPs) and related controller synthesis methods, while also emphasizing the connection between MDPs and the probabilistic models of Markov Chains.

2.1.1 Markov Chains

Definition 1 (Markov Chains)**.** *A Markov Chain is defined by tuple $M = (S, s_I, P)$ where S is a finite set of states, $s_I \in S$ is initial state and $P : S \times S$ is the transition probability matrix where $\sum_{s' \in S} P(s, s') = 1$.*

A Markov chain is a simple state space model composed of states and probabilistic transitions between them. An agent in a Markov chain explores the state space, transitioning from one state to another or remaining in the same state according to the transition probabilities. The sum of transition probabilities from each state to all other states, including itself, is always equal to one. By definition, a Markov chain possesses the Markov property, which states that the future behavior of a stochastic process depends only on its current state and not on its past history [15].

Example 1. *In figure 2.1 we can see classical problem introduced by Knuth and Yao [11], where they use a Markov chain to model a 6-sided die using coin flipping. Starting from initial state $s0$, one has probability of 0.5 (heads) of transitioning to state $s1$ and probability of 0.5 (tails) of transitioning to state $s2$. Every following state in middle layers states* *the same, ultimately resulting in terminating states labeled 1 to 6. In span of few steps, probability of reaching each terminating state is equal to 1/6.*

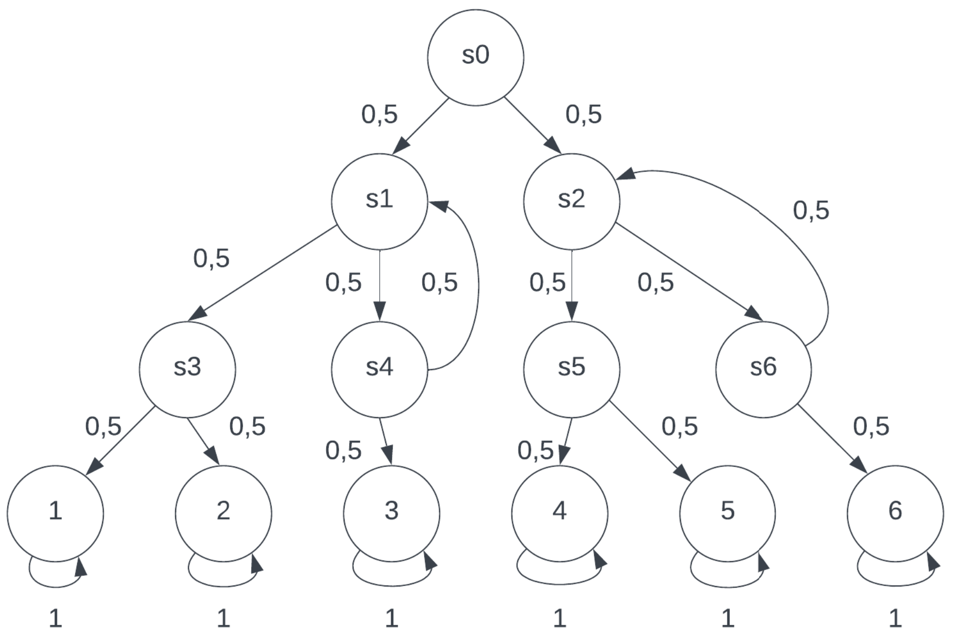

Figure 2.1: Markov chain simulating six sided die

2.1.2 Markov Decision Processes

While Markov chains provide a useful tool for modeling stochastic processes with the Markovian property, they lack the ability to include decision-making and goal-directed behavior, which is reason, why Markov chains are extended by Markov decision processes (MDPs).

Definition 2 (Markov Decision Process)**.** *MDP is a tuple $M = (S, s_I, A, P_a, R_a)$, where S is finite set of states, $s_I \in S$ is initial state, A is set of actions, $P_a(s, s')$ is probability of transitioning from state s to state s' when action a is taken, and $R_a(s, s')$ is immediate reward received upon transitioning from state s to state s'.*

Markov Decision Processes (MDPs) represent an advancement over Markov chains by incorporating actions into the state-transition model. In MDPs, an agent, which could be a real or hypothetical entity, performs actions to interact with the modeled system. Each action results in a probability distribution over a subset of possible states, influencing the transitions between states. In an MDP, the ultimate goal is usually to find an optimal policy that can guide the decision-making of an agent within the model to optimize a specified objective, such as maximizing the expected cumulative reward over time, minimizing a cost function, or satisfying certain constraints.

2.1.3 Value Function for MDP

The value function serves as a versatile tool for analyzing and resolving MDPs. The value function assigns a real number to each state within the state space, which corresponds to the anticipated cumulative reward an agent can achieve by following a specific policy from a given state [12, p. 136].

Definition 3 (Value Function)**.** *Formal specification of value function is:*

$$V(s) = \max_{a \in A} \left[R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V(s') \right],$$

where $s \in S$ is state from the state space, $a \in A$ represent action from the action space, $R(s,a)$ is immediate reward received upon taking action a from state s, $P(s'|s,a)$ is transition probability of transition into state s' from state s after taking action a and γ is discount factor, $0 \leq \gamma < 1$.

The value function provides a measure of the desirability of each state, as it quantifies the long-term reward the agent can expect by following the best possible course of action. As such, it serves as a foundation for determining optimal policies in MDPs, as the agent should choose actions that lead to states with higher values.

To find the optimal value function and the corresponding optimal policy, dynamic programming algorithms such as value iteration and policy iteration are often utilized. These algorithms use Bellman equation, a recursive relationship that the value function must satisfy, to iteratively converge to the optimal solution, which we will discuss in next section.

2.1.4 Bellman Equation

The Bellman equation has strong connection with problem of determining an optimal policy of an MDP. It defines a recursive relationship of the value function of a specific state and the value functions of the following states. The value function represents the expected cumulative reward that an agent can obtain from a given state while following a specific policy. The Bellman equation is the main cornerstone in dynamic programming algorithms like value iteration and policy iteration and is essential for calculating policies [12, p. 142].

Definition 4 (Bellman equation)**.** *In its general form, Bellman equation is defined as:*

$$V^{\pi}(s) = \sum_{a \in A} \pi(a|s) \sum_{s' \in S} P(s'|s,a) \left[R(s'|s,a) + \gamma V^{\pi}(s') \right]$$

where $V^{\pi}(s)$ is the value function of state s under policy π, $\pi(a|s)$ is the probability of taking action a in state s under policy π, $P_a(s,s')$ is the transition probability from state s to state s' when taking action a, $R_a(s,s')$ is the immediate reward received when transitioning from state s to state s' after taking action a, γ is the discount factor and $V^{\pi}(s')$ is the value function of state s' under policy π - the recursive part of equation.

2.1.5 Value Iteration

Having discussed value function and Bellman equation, we can now delve into the issue of searching for optimal solutions of MDPs. Value iteration is a dynamic programming algorithm used to find the optimal value function and policy for an MDP [12, p. 141].

It iteratively updates the value function using the Bellman equation until convergence, at which point the optimal value function is obtained. The update equation for value iteration is:

Definition 5 (Value Iteration)**.**

$$V^{(i+1)}(s) = \max_a \left[R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^{(i)}(s') \right], 0 \leq \gamma < 1$$

where $V^{(i+1)}(s)$ denotes Value function of state $s \in S$ in iteration $i+1$, $R(s,a)$ is reward obtained when taking action $a \in A$ in state s, γ is discount factor, $P(s'|s,a)$ is probability of transfering to state s' from state s upon taking action a.

Algorithm 1 MDP Value Iteration Algorithm

```
1: procedure MDPValueIteration($S, A, P, R, \gamma, \epsilon$)
2:     Initialize $V^{(0)}(s) \leftarrow 0$ for all $s \in S$
3:     $k \leftarrow 0$
4:     while not converged do
5:         for each $s \in S$ do
6:             $V^{(k+1)}(s) \leftarrow \max_{a \in A} \left[R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^{(k)}(s')\right]$
7:         end for
8:         if $\max_{s \in S} |V^{(k+1)}(s) - V^{(k)}(s)| < \epsilon$ then
9:             converged $\leftarrow$ True
10:        else
11:            $k \leftarrow k+1$
12:        end if
13:    end while
14:    for each $s \in S$ do
15:        $\pi^*(s) \leftarrow \arg\max_{a \in A} \left[R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^*(s')\right]$
16:    end for
17:    return $\pi^*, V^*$
18: end procedure
```

For each state in model, in each iteration of value iteration, value function is updated based on the maximum of the sum of the immediate reward for taking an action from that state and the discounted value function of the successor states. This process continues iteratively, until conditions of value iteration are met. Typically, the goal is to achieve convergence of the value function, which is recognized when a desired level of accuracy, denoted by ϵ, is reached.

Example 2. *Let's consider simple Markov decision process on figure 2.2 with discount factor $\gamma = 0.9$ and $\epsilon = 0.2$.*

To calculate first iteration of value iteration, we initialize value function for each state as 0. Then, we iteratively apply value iteration equation for each state. Let's compute first iteration of value iteration.

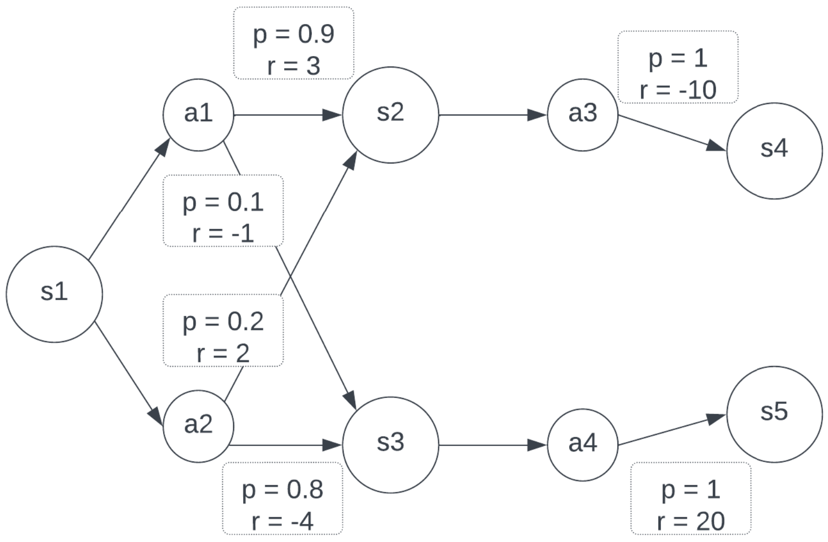

Figure 2.2: Simple Markov decision process

$$V^{(1)}(s1) = \max_a [3 + 0.9 \cdot (0.9 \cdot 0 + 0.1 \cdot 0), -1 + 0.9 \cdot (0.2 \cdot 0 + 0.8 \cdot 0)] = 3.0 \tag{2.1}$$
$$V^{(1)}(s2) = \max_a [-10 + 0.9 \cdot (1 \cdot 0)] = -10.0 \tag{2.2}$$
$$V^{(1)}(s3) = \max_a [20 + 0.9 \cdot (1 \cdot 0)] = 20.0 \tag{2.3}$$

Value function for states $s4$ and $s5$ remain 0. Now let's continue with second iteration.

$$V^{(2)}(s1) = \max_a [3 + 0.9 \cdot (0.9 \cdot (-10) + 0.1 \cdot (20)), -1 + 0.9 \cdot (0.2 \cdot (-10) + 0.8 \cdot 20)] = 11.6 \tag{2.4}$$
$$V^{(2)}(s2) = \max_a [-10 + 0.9 \cdot (1 \cdot 0)] = -10.0 \tag{2.5}$$
$$V^{(2)}(s3) = \max_a [20 + 0.9 \cdot (1 \cdot 0)] = 20.0 \tag{2.6}$$

As we can see, value function for $s1$ has been updated, while the others have remained unchanged. In the table below (Table 2.1), it is evident that after 5 iterations, the convergence criterion is satisfied. For every state, the difference between the value functions for iteration 5 and the previous iteration 4 is less than the specified epsilon (0.2).

Iteration	$V(s_1)$	$V(s_2)$	$V(s_3)$	$V(s_4)$	$V(s_5)$
0	0	0	0	0	0
1	3.0	-10	20	0	0
2	11.6	-10	20	0	0
3	14.34	-10	20	0	0
4	14.706	-10	20	0	0
5	14.7354	-10	20	0	0

Table 2.1: Table 1: First five iterations of value iteration over MDP on Figure 2.2

2.2 Partially Observable Markov Decision Processes

Until now, we have not adressed uncertainty in the sense of imperfect information about observed system. In Markov decision process, agent always has perfect knowledge about system and its state, and based on trained policy, we can accurately predict agent's next move. To fully incorporate uncertainty into our model, we will use Partially observable Markov decision processes - POMDPs. Partially observable Markov decision processes extend MDPs by addition of observation models, reflecting the uncertainty in perception of the environment's true state [2].

Definition 6 (Partially Observable Markov Decision Process)**.** *POMDP is a tuple $\mathcal{M} = (M, Z, O)$, where the M is Markov decision process being extended, Z is finite set of observations, O is (optionally deterministic) observation function that returns observation $O(s) = z \in Z$ for every state s. The observation $z \in Z$ is trivial, if there is only one state $s \in S$ with $O(s) = z$.*

2.2.1 Example - Crying Baby Problem

To better illustrate this concept, let's examine classic example of Partially Observable Markov Decision Process called „Problem of crying baby“, described in the book Algorithms for Decision Making [12, p. 382]. Problem is defined as following:

Example 3 (Crying Baby Problem)**.** *An agent is taking care of a baby by choosing whether to feed the baby at each timestep. Initially, the baby is hungry. When the agent chooses to feed the baby, the probability of the baby becoming full is 100 %. If the baby is hungry and the agent does not feed it, the baby stays hungry with a 100 % probability. When the baby is full, there is a 10 % probability of it becoming hungry in the next timestep. However, the current state of the baby cannot be precisely known. The baby cries with a 10 % probability when full, whereas a hungry baby cries with an 80 % probability. When baby becomes hungry, agent receives reward of -10, and when baby is fed, agent receives reward of -5. Discount factor is* 0.9.

From this description, we can extract formal specification of described POMDP.

- ***States:*** $S = \{Hungry, Full\}$
- ***Actions:*** $A = \{Feed, Don't\ Feed\}$
- ***Transition probabilities:***
  - $P(Hungry|Hungry, Feed) = 0$

  - $P(Full|Hungry, Feed) = 1$
  - $P(Hungry|Hungry, Don't\ Feed) = 1$
  - $P(Full|Hungry, Don't\ Feed) = 0$
  - $P(Hungry|Full, Feed) = 0$
  - $P(Full|Full, Feed) = 1$
  - $P(Hungry|Full, Don't\ Feed) = 0.1$
  - $P(Full|Full, Don't\ Feed) = 0.9$

- ***Rewards:***
  - $R(Feed) = -5$
  - $R(Hungry) = -10$

- ***Observations:*** $Z = \{Crying, Not\ Crying\}$

- ***Observation probabilities:***
  - $P(Crying|Hungry) = 0.8$
  - $P(Not\ Crying|Hungry) = 0.2$
  - $P(Crying|Full) = 0.1$
  - $P(Not\ Crying|Full) = 0.9$

- ***Discount factor:*** $\gamma = 0.9$

This formal specification can be displayed as follows in the figure 2.3, where the arrows symbolize that next state (s_{t+1}) and reward (r_t) depends on current state (s_t) and action (a_t) taken in current timestamp, and observation (o_t) depends on current state.

2.2.2 Beliefs and Observations

A belief state is a probability distribution over the possible states in the environment, representing the agent's current knowledge about the true state of the system. As the agent interacts with the environment, it updates its belief state based on the actions it takes and the observations it receives.

Observations in POMDPs serve as the agent's incomplete and potentially noisy information about the true state of the environment, which is used to model the uncertainty inherent in these models. Unlike MDPs, the agent does not have complete knowledge of the current state, relying instead on the gathered observations.

Belief-based algorithms, such as SARSOP, present methods for solving Partially Observable Markov Decision Processes. These algorithms leverage the concept of belief states, which encapsulate the agent's knowledge about the system state, providing a principled approach to handle uncertainties. SARSOP, in particular, distinguishes itself by focusing on the optimally reachable subset of the belief space, effectively reducing the computational complexity [13]. However, this approach was proven less efective than inductive synthesis [2], which we will discuss later in 3.

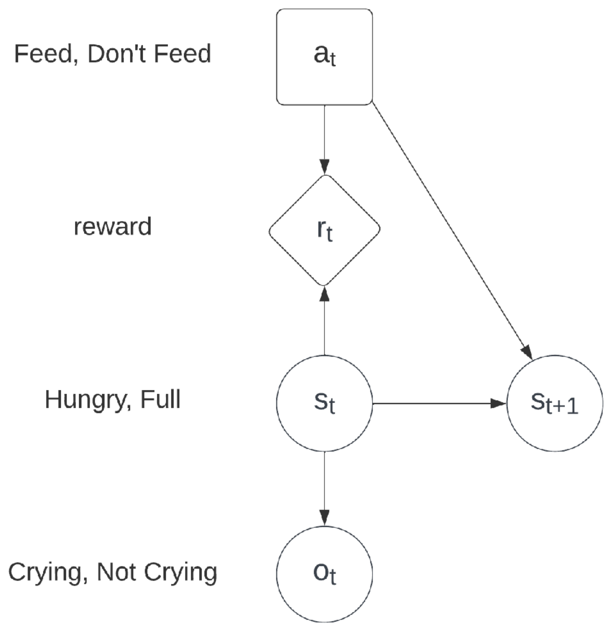

Figure 2.3: Partially Observable MDP modelling the Crying Baby Problem

2.2.3 The Bayesian Update Rule

The agent updates its belief about the state of model using the observation function, taking into account the actions performed and the observations received. To better understand this process, let's consider the Bayesian update rule, taken from the book Algorithm for Decision Making [12, p. 408], that governs the transition of belief states in a POMDP:

$$\begin{aligned}
T(b'|b,a) &= P(b'|b,a) &(2.7)\\
&= \sum_o P(b'|b,a,o)P(o|b,a) &(2.8)\\
&= \sum_o P(b'|b,a,o)\sum_s P(o|b,a,s)P(s|b,a) &(2.9)\\
&= \sum_o P(b'|b,a,o)\sum_s P(o|b,a,s)b(s) &(2.10)\\
&= \sum_o \left[b' = \text{Update}(b,a,o)\right]\sum_s O(o|a,s')\sum_{s'} T(s'|s,a)b(s) &(2.11)
\end{aligned}$$

This equation represents the belief update process, which computes the probability of the next belief state, b', given the current belief state, b, the action taken, a, and the observation received, o. The update rule incorporates both the observation function and the transition function, ensuring that the agent's belief state is updated accordingly as it interacts with the environment.

2.2.4 Bayes-Adaptive Markov Decision Process

Next I want to briefly mention Bayesian Adaptive MDPs as an example of transforming a POMDP with continuous belief states into a discrete representation, which can facilitate the development of more efficient algorithms and decision-making strategies.

In the book Algorithms for Decision Making, it is stated that: „We can formulate the problem of acting optimally in an MDP with an unknown model as a higher-dimensional MDP with a known model. This MDP is known as a Bayes-adaptive Markov decision process“ [12, p. 329].

In other words, we can transform a POMDP, which features an agent with uncertain knowledge of its true state, into an equivalent MDP with an enlarged state space. By making this transformation, we can utilize existing MDP solutions and techniques while preserving uncertainty inherent in the model.

The state space of such MDP is product of $S \times B$, where S is discrete state space of previous POMDP and B is the belief state space, so the state is a pair (s, b), $s \in S$ and $b \in B$.

Transition in Bayes-adaptive MDP is defined as $T(s', b' | s, b, a)$ – probability of transitioning to state s' with a belief state b' assuming agent's initial state s with belief state b when agent chooses action a.

2.3 POMDP Solving Methods

Finding optimal policies of partially observable MDPs can be exceptionally challenging due to the uncertainty of agent's knowledge of true state of environment and belief being part of continuous space, rather than discrete state space of MDPs. In addition, the problem of finding optimal policies is generally undecidable, meaning there is no algorithm that will determine, for all inputs, whether the problem has a solution [2]. Various approaches can be taken when searching for optimal policy, including exact methods, approximate methods, online methods or reinforcement learning approaches. Because my work is based on exact methods, I will mainly focus on them. This section of the document was primarily grounded in the concepts presented in [5].

2.3.1 POMDP-specific Value Iteration

Previously, we discussed the Value Iteration algorithm for solving Markov Decision Processes. In this chapter, we will explore the Value Iteration algorithm for Partially Observable Markov Decision Processes, from which we will move onto another exact methods.

While the core idea of the Value Iteration algorithm remains the same for both MDPs and POMDPs, there are key differences that arise due to the partial observability of the environment in POMDPs. In MDP Value Iteration, we operate directly on the state space, whereas in POMDP Value Iteration, we work with the belief space, which is a continuous representation of the agent's uncertainty about the environment.

Another important distinction is the introduction of an observation model that governs the probability of receiving specific observations given the agent's belief state and action.

Algorithm 2 POMDP Value Iteration Algorithm

```
1: procedure POMDPValueIteration($S, A, O, P, R, Z, \gamma, \epsilon$)
2:     Initialize $V^{(0)}(b) \leftarrow 0$ for all $b \in B$ (belief space)
3:     $k \leftarrow 0$
4:     while not converged do
5:         for each $b \in B$ do
6:             $V^{(k+1)}(b) \leftarrow \max_{a\in A} \left[R(b,a) + \gamma \sum_{o\in O} Z(o|b,a) \sum_{s'\in S} P(s'|b,a)V^{(k)}(b')\right]$
7:         end for
8:         if $\max_{b\in B} |V^{(k+1)}(b) - V^{(k)}(b)| < \epsilon$ then
9:             converged $\leftarrow$ True
10:        else
11:            $k \leftarrow k + 1$
12:        end if
13:    end while
14:    for each $b \in B$ do
15:        $\pi^*(b) \leftarrow \arg\max_{a\in A} \left[R(b,a) + \gamma \sum_{o\in O} Z(o|b,a) \sum_{s'\in S} P(s'|b,a)V^*(b')\right]$
16:    end for
17:    return $\pi^*, V^*$
18: end procedure
```

In the POMDP Value Iteration algorithm, the main differences from the MDP Value Iteration are the use of belief states $b \in B$ and the observation model $Z(o|b,a)$. B is the belief space, which represents probability distributions over the underlying state space S, $V^{(k)}(b)$ is the value function at iteration k for a belief state b, $R(b,a)$ is the expected immediate reward for taking action a in belief state b. $P(s'|b,a)$ is the probability of transitioning to state s' given belief state b and action a, $Z(o|b,a)$ is the probability of observing o given belief state b and action a, $V^*(b')$ is the optimal value function for belief state b' and $\pi^*(b)$ is the optimal action to take in belief state b.

The algorithm starts by initializing the value function for all belief states to zero. Then, it enters a loop that continues until convergence is achieved (i.e., the maximum change in the value function is less than the specified threshold ϵ).

In each iteration, the algorithm updates the value function for each belief state by taking the maximum over all actions of the sum of the immediate reward and the expected future rewards, considering both the state transition probabilities and observation probabilities. After convergence, the optimal policy is computed by selecting the action that maximizes the sum of immediate and expected future rewards for each belief state.

The key difference between the MDP and POMDP Value Iteration algorithms is that the POMDP version operates on belief states instead of directly on the states of the environment. This allows it to account for the inherent uncertainty in the POMDP setting.

2.3.2 Policy Iteration

Policy Iteration is another powerful technique for solving both Markov Decision Processes (MDPs) and Partially Observable Markov Decision Processes (POMDPs). While the previously discussed Value Iteration method focuses on iteratively updating the value function until convergence, Policy Iteration takes a different approach by alternately improving the policy and evaluating it. This approach can often lead to faster convergence, as it directly optimizes the policy rather than the value function.

The Policy Iteration algorithm for solving MDPs and POMDPs consists of two main steps: policy evaluation and policy improvement. The algorithm starts by initializing an arbitrary policy $\pi^{(0)}$ and setting an iteration counter k to zero. The algorithm then proceeds iteratively until convergence is achieved:

Algorithm 3 Policy Iteration Algorithm

```
 1: procedure PolicyIteration($S, A, P, R, \gamma$)
 2:     Initialize an arbitrary policy $\pi^{(0)}$
 3:     $k \leftarrow 0$
 4:     while not converged do
 5:         Perform policy evaluation to compute $V^{\pi^{(k)}}$
 6:         for each $s \in S$ do
 7:             $\pi^{(k+1)}(s) \leftarrow \arg\max_{a \in A} \left[R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^{\pi^{(k)}}(s')\right]$
 8:         end for
 9:         if $\pi^{(k+1)} = \pi^{(k)}$ then
10:             converged $\leftarrow$ True
11:         else
12:             $k \leftarrow k + 1$
13:         end if
14:     end while
15:     return $\pi^{(k)}, V^{\pi^{(k)}}$
16: end procedure
```

Let us walk through this algorithm:

Policy Evaluation: In this step, the algorithm computes the value function $V^{\pi^{(k)}}$ for the current policy $\pi^{(k)}$. This can be done using various methods, such as solving a system of linear equations, dynamic programming, or iterative methods like the Bellman Expectation Equation. The goal of this step is to obtain an accurate estimate of the value function for the current policy, which will be used to improve the policy in the next step.

Policy Improvement: Once the value function has been computed, the algorithm proceeds to update the policy by selecting the action that maximizes the expected return at each state. This is achieved by iterating through all states $s \in S$ and choosing the action $a \in A$ that maximizes the following expression: $R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^{\pi^{(k)}}(s')$. The updated policy $\pi^{(k+1)}$ is then obtained by assigning the chosen action to each state.

The algorithm checks for convergence by comparing the current policy $\pi^{(k+1)}$ with the previous policy $\pi^{(k)}$. If the policies are identical, the algorithm is considered to have converged, and the final policy $\pi^{(k)}$ and its associated value function $V^{\pi^{(k)}}$ are returned. Otherwise, the iteration counter k is incremented, and the process is repeated from the policy evaluation step.

One of the key advantages of Policy Iteration is that it often converges faster than Value Iteration, as it directly optimizes the policy rather than the value function. However, it may require solving more complex systems of equations during the policy evaluation step, which can be computationally expensive for large or continuous state spaces.

2.3.3 Point Based Value Iteration

Point-based Value Iteration (PBVI) is a approximate algorithm for solving Partially Observable Markov Decision Processes, which addresses the challenges posed by continuous belief spaces by focusing on a finite set of representative belief points. PBVI aims to establish a balance between computational efficiency and solution quality by selectively updating the value function only at these representative points.

Algorithm 4 Point-Based Value Iteration

```
Require: POMDP model $(S, A, O, T, R, \Omega, \gamma)$, set of belief points $B$, max iterations $N$
1: Initialize $V^0 = \{\alpha_1, \alpha_2, \ldots, \alpha_{|S|}\}$ where $\alpha_i(s) = R(s, a_i)$
2: for $n = 1$ to $N$ do
3:     $V^n \leftarrow \emptyset$
4:     for each $b \in B$ do
5:         $\alpha_b^* \leftarrow \underset{\alpha \in \Gamma(V^{n-1})}{\operatorname{argmax}} \sum_{s \in S} b(s) \cdot \alpha(s)$
6:         Add $\alpha_b^*$ to $V^n$
7:     end for
8: end for
9: Derive the final policy $\pi$ from $V^N$
```

In the Point-Based Value Iteration (PBVI) algorithm, the focus is on a finite set of representative belief points B, instead of updating the value function for the entire continuous belief space. This significantly reduces the computational complexity of the algorithm, making it more efficient than Value Iteration (VI) and Policy Iteration (PI) when dealing with large or continuous belief spaces.

The PBVI algorithm begins by initializing the value function V^0 with α-vectors corresponding to the immediate rewards for each action. Then, for a given number of iterations N, the algorithm iteratively updates the value function by selecting the best α-vector for each belief point $b \in B$ from the set $\Gamma(V^{n-1})$, which is the cross-sum of the α-vectors in the previous iteration. Once the iterations are complete, the final policy π is derived from the resulting value function V^N.

The primary advantage of PBVI over VI and PI is its computational efficiency, as it selectively updates the value function at a finite set of belief points rather than the entire belief space. This allows the algorithm to scale better to large POMDPs with high-dimensional or continuous belief spaces.

However, it is important to note that PBVI is an approximate algorithm and may not always find the optimal policy. The quality of the solution depends on the choice of representative belief points and the number of iterations.

2.4 Conclusion

In this chapter, we have delved into the principles of Markov Decision Processes (MDPs) and their extension, Partially Observable Markov Decision Processes (POMDPs). These serve as pivotal tools for decision-making optimization problems under uncertainty. MDPs enable the modeling of complex systems and the computation of optimal policies via algorithms like value iteration and policy iteration, which exploit the recursive nature of the Bellman equation.

We introduced POMDPs as an advanced layer, adding observations to represent the agent's incomplete knowledge of the environment. We discussed belief states and the Bayesian update rule for managing belief state transitions, capturing decision-making dynamics within a probabilistic framework.

Finally, we explored three key POMDP-solving algorithms: Value Iteration, Policy Iteration, and Point-Based Value Iteration. While the first two offer exact methods, they may be computationally demanding for larger belief spaces. To mitigate this, Point-Based Value Iteration provides a near-optimal solution with reduced computational complexity, demonstrating diverse strategies to tackle decision-making under uncertainty.

Chapter 3

Inductive Synthesis

Partially Observable Markov Decision Processes (POMDPs) pose significant challenges when it comes to finding optimal policies, mainly due to the inherent uncertainty and the continuous nature of belief states. Finite State Controllers (FSCs) and Inductive Synthesis are two closely related concepts that have emerged as promising approaches for addressing these challenges and synthesizing policies for POMDPs.

In this chapter, we will first introduce the concept of Finite State Controllers, a family of memory structures that can represent policies for POMDPs, and discuss their properties and advantages. Next, we will delve into the idea of Inductive Synthesis, a technique for learning FSCs with the goal of reducing memory requirements and computational complexity while maintaining the desired level of performance. We will introduce Abstraction Refinement concept and mention probabilistic programming.

The theoretical framework and methodologies discussed in this chapter are largely drawn from the research presented in „Inductive Synthesis of Finite-State Controllers for POMDPs“ by R. Andriushchenko, M. Češka et al. [2].

3.1 Finite State Controllers

A Finite State Controller (FSC) serves as a compact mechanism to represent policies or control strategies for an agent operating in partially observable environments, like Partially Observable Markov Decision Processes. FSCs are structured as directed graphs, where nodes symbolize controller states and edges denote transitions based on actions and observations.

Finite State Controllers can be categorized as Moore machines or Mealy machines. In the context of Moore machines, an action is associated with each node (or controller state), whereas for Mealy machines, the action is determined by the transitions. This work particularly focuses on Mealy machines.

Upon receiving an observation, the agent transitions from its current controller state to a new one via the corresponding edge, executing the action linked to the new controller state. The FSC provides a deterministic framework for decision-making, which is computationally efficient, especially when dealing with large or continuous state spaces.

FSCs are integral to various POMDP solution methods, such as policy iteration, point-based value iteration, and reinforcement learning algorithms, among others. The primary objective of these algorithms is to identify an FSC that maximizes the agent's expected cumulative reward over time, given the constraints of operating in a partially observable environment.

Finite State Controllers and belief-based approaches, like SARSOP, provide distinct strategies to handle the complexities of decision-making within Partially Observable Markov Decision Processes (POMDPs), with FSCs offering a deterministic and computationally efficient approach, while belief-based methods like SARSOP provide a principled and direct way to handle uncertainty. However, in the context of efficient model checking, FSC are better suited for the role, as they offer more compact representation of policies [2].

Definition 7 (Finite State Controller)**.** *A Finite State Controller (FSC) for a POMDP can be formally described as a tuple: $FSC = (N, n_0, \gamma, \delta)$ where N is a finite set of nodes (controller states), $n_0 \in N$ is the initial node, $\gamma : N \times Z \to A$ is a function that determines the action when the agent is in node n and observes z, and $\delta : N \times Z \to N$ is a function that updates the memory node to $\delta(n, z)$, when the agent is in node n and observes z.*

In an FSC, when the agent in node n observes z, it executes the action determined by $\gamma(n, z)$ and transitions to the next node $\delta(n, z)$ as per the function δ. This process aligns with the description of the FSC as a Mealy machine, where the output (action) is determined by the transition taken based on the observation.

3.1.1 Families of Finite State Controllers

When considering decision-making strategies in Partially Observable Markov Decision Processes, it is valuable to explore different structures of Finite State Controllers which can be grouped into collections, referred to as families of FSCs. These families represent various configurations of FSCs with respect to their memory usage, providing a versatile tool for investigating diverse policy representations.

A family of FSCs corresponds to a set of FSCs that are induced by a POMDP, yielding a set of Markov Chains. These FSCs are categorized into two main types: full and reduced FSCs, based on the number of memory nodes and their usage with observations.

A full k-FSC family, denoted as $F_k = (N, n_0, K)$, comprises k nodes (N), an initial node (n_0), and a finite set of parameters ($K = N \times Z$) each having a domain $V(n, z) \subseteq \text{Act} \times N$. Each parameter helps determine the action $\gamma(n, z)$ and the next node $\delta(n, z)$. In other words, a family of full k-FSCs defines a set of FSCs varying by the substitutions of the parameters.

On the other hand, a reduced family F_μ, characterized by a memory model $\mu : Z \to N$, is a sub-family of F_k for $k = \max_{z \in Z} \mu(z)$. In this case, the number of memory nodes used in the observation z is given by $\mu(z)$. This form of family induces a smaller design space and requires less memory, which can aid in interpretability.

3.1.2 Constraints and Optimization Specifications

Specifications are a useful tool that allow us to formally define the desirable properties and goals of our system, thereby guiding the decision-making process of the agent. By defining constraints as quantitative properties, we can impose specific requirements on the system's behavior.

These constraints can be indefinite-horizon reachability properties or expected reward properties, with thresholds serving as the quantitative benchmarks that the system is expected to achieve. Additionally, we also define an optimization objective, which is the primary goal that the system aims to maximize or minimize. This can be either reachability probabilities or expected reward properties. Together, the constraints and optimization objective shape the behavior of the FSC, driving it towards the desired outcomes while respecting the imposed limitations.

3.2 Inductive Synthesis

This overview of Inductive Synthesis is based on work of Andriushchenko, Češka et al. [2]. Inductive Synthesis is a technique originally developed in the context of program synthesis, where the goal is to construct a program that provably satisfies a given formal specification. Inductive Synthesis can be used to learn an optimal or near-optimal FSC with reduced memory requirements and computational complexity.

The Inductive Synthesis framework for FSC learning consists of two main stages: an outer stage, where the learner constructs a design space containing finitely many FSCs, and an inner stage, where the teacher provides the best FSC within the design space, along with potentially additional diagnostic information.

The learner begins with a small design space and strategically modifies it based on the feedback obtained from the teacher. The teacher, on the other hand, determines the best FSC within the design space using various methods, such as enumeration, branch-and-bound [9], or mixed-integer linear programming (MILP) [1].

Both the learner and teacher have access to an additional oracle that over-approximates the design space. This abstract design space can be efficiently analyzed as it resembles the analysis of fully observable policies. The oracle provides constraints on what the best FSC within the original design space can achieve, which is essential for guiding the search in both stages.

3.3 Counterexample-Guided Inductive Synthesis

The theoretical foundations for this part of work were primarily drawn from the work of [6].The central idea of Counterexample-Guided Inductive Synthesis (CEGIS) is to iteratively refine the design space by using counterexamples. These counterexamples, which are generated from failed attempts at satisfying the specification, provide essential feedback that helps to constrain and guide the synthesis process. This iterative process continues until an FSC is found that meets the desired specifications, or until it is proven that no such FSC exists within the defined design space.

The following image 3.1 represents principle of Counterexample-guided Inductive Synthesis. In this context, a sketch represents a space of potential solutions, with certain parts left unspecified. As we'll explore later in this chapter, CEGIS progressively prunes this solution space by iteratively refining the sketch based on counterexamples from failed verification attempts. This process continues until a solution satisfying the given specification is found or until all possible solutions have been examined.

The following algorithm 5 represents an adaptation of the Counterexample-Guided Inductive Synthesis algorithm, as detailed in [3]. This revised version is designed to work with a family of FSCs, as opposed to the original version which works with a family of Markov Chains. The objective remains the same: to discover a realization of the FSC that fulfills a specified reachability property, or alternatively, to return UNSAT if no such realization can be found.

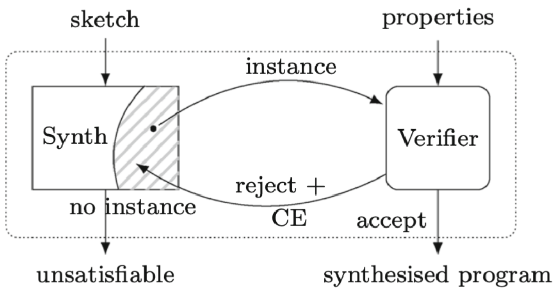

Figure 3.1: Principle of CEGIS procedure. Image has been adapted from the paper by Češka et al. [7].

Algorithm 5 Counterexample-guided Inductive Synthesis for FSCs

```
Require: A family $\mathcal{F}_k = (N, n_0, K)$ of FSCs, a reachability property $\varphi$.
Ensure: Realization $r \in R\mathcal{F}_k$ such that $\mathcal{F}_{k,r} \models \varphi$, or UNSAT if no such realization exists.
 1: procedure CEGIS($\mathcal{F}_k = (N, n_0, K), \varphi$)
 2:     $\psi \leftarrow$ Initialize($\mathcal{F}_k$)
 3:     $r \leftarrow$ GetRealization($\psi$)
 4:     while $r \neq \emptyset$ do
 5:         $C \leftarrow$ Verify($\mathcal{F}_{k,r}, r, \varphi$)
 6:         if $C$ then
 7:             return $r$
 8:         end if
 9:         $\psi \leftarrow \psi \wedge \left(\bigwedge_{r \in C}\right.$ LearnFromConflict($\left.\mathcal{F}_{k,r}, r)\right)$
10:         $r \leftarrow$ GetRealization($\psi$)
11:     end while
12:     return UNSAT
13: end procedure
```

The algorithm takes as input a family $\mathcal{F}_k = (N, n_0, K)$ of FSCs and a reachability property φ. The family of FSCs is composed of a set of nodes N, an initial node n_0, and a finite set of parameters $K = N \times Z$. Each parameter helps determine the action $\gamma(n, z)$ and the next node $\delta(n, z)$. The reachability property φ is a formal specification that the FSC should satisfy.

CEGIS algorithm outputs a realization $r \in R\mathcal{F}k$ such that the FSC $\mathcal{F}k, r$ satisfies the reachability property φ. If no such realization exists, the algorithm returns UNSAT.

The function Initialize takes the family of FSCs $\mathcal{F}_k$ as an input and returns an initial set of realizations.

The GetRealization function takes the set of realizations and returns a particular realization r. This can be done in any manner, for example, by choosing a random realization from the set.

The Verify function takes the family of FSCs $\mathcal{F}k$, a realization r, and the reachability property φ as inputs. It checks whether the FSC $\mathcal{F}k, r$ satisfies the reachability property φ. If it does, it returns None. Otherwise, it returns a counterexample, which is a subsystem of the FSC that violates the reachability property.

The LearnFromConflict function takes the family of FSCs $\mathcal{F}_k$ and a counterexample as inputs. It returns a set of constraints that exclude the invalid realization from the set of realizations. These newly generated constraints, when combined with the existing ones, prevent the algorithm from considering similar invalid realizations in subsequent iterations, thereby refining the search space towards valid solutions.

3.4 Counterexample-Guided Abstraction Refinement

Counterexample-Guided Abstraction Refinement (CEGAR) is a powerful technique in the field of formal verification [8]. It is designed to increase the efficiency of system verification by working with simplified or abstracted versions of the system model. The key idea behind CEGAR is to iterate between two phases: abstraction, where the system model is simplified, and verification, where the simplified model is checked against a specific specification. If the model fails to meet the specification, a counterexample is produced that guides the refinement of the abstraction in the next iteration. Through this iterative process, CEGAR allows for the verification of complex systems without having to examine every single detail of the system in every iteration.

CEGAR is particularly beneficial when dealing with large and complex systems where traditional verification methods would be computationally expensive or practically infeasible. By focusing on abstracted models and only refining these models when necessary, CEGAR significantly reduces the computational burden. This technique is widely used in software model checking, hardware verification, and control synthesis.

3.5 Probabilistic Programs

Probabilistic programs are an advanced modelling framework utilized to depict systems that include elements of stochastic uncertainty. They aim to satisfy a set of temporal constraints that define their correctness and efficiency. During the early stages of system design, these programs often remain incomplete or contain 'holes', representing undefined or partially implemented components. This necessitates a process called design space exploration, which involves analyzing and filling these holes with appropriate behaviors or subsystems. A significant challenge in this process is to efficiently represent and reason about various possible designs or 'realizations'. To address this, the concept of 'sketching' is often employed, providing a concise representation of the family of potential designs. This section is based on the theory and methodology outlined in Paynt: A Tool for Inductive Synthesis of Probabilistic Programs by Andriushchenko, Češka et al. [4].

3.6 Conclusion

In this chapter, we have established the foundational concepts necessary to comprehend the nuances of inductive synthesis. Our discussion commenced with an exploration of finite-state controllers, elaborating on their family structures, constraints, and optimization objectives. Subsequently, we ventured into the problematics of inductive synthesis, with a specific focus on the Counterexample-Guided Inductive Synthesis (CEGIS) and Counterexample-Guided Abstraction Refinement (CEGAR) methodologies.

Furthermore, we introduced the concept of probabilistic programs, a key aspect in understanding the broader context of formal verification. This discussion served as a stepping stone to our examination of specific examples of formal model checking tools, namely Prism, Storm, and Paynt. Through this concise overview, we have paved the way for deeper exploration and application of these essential tools and techniques in next chapter.

Chapter 4

Exploring Formal Methods Tools: PRISM, STORM and PAYNT

Having laid the theoretical groundwork in the preceding chapters, we are now well prepared to delve into the practical nuances of model checking. In this chapter, we will focus on some of the most prominent tools in this domain: PRISM, a renowned tool for formal modeling and analysis; STORM, a formidable stochastic model checker; and PAYNT, a probabilistic program synthesizer. The optimization of PAYNT, in particular, will be the main focus of our discussion, as it forms the primary objective of this work.

4.1 PRISM

The information presented in this section has been gathered from the following sources [16, 14]. According to the official PRISM model checker manual [16]: „PRISM is a probabilistic model checker, a tool for formal modelling and analysis of systems that exhibit random or probabilistic behaviour. It has been used to analyse systems from many different application domains, including communication and multimedia protocols, randomised distributed algorithms, security protocols, biological systems and many others.“

PRISM is a versatile probabilistic model checker that supports a wide variety of probabilistic models, including discrete-time Markov chains (DTMCs), continuous-time Markov chains (CTMCs), Markov decision processes (MDPs) or Partially observable Markov decision processes (POMDPs). These models are defined using the PRISM language, which is a state-based language designed for simplicity and effectiveness.

PRISM allows for the automated analysis of numerous quantitative properties related to these models, such as failure probabilities within specified timeframes, worst-case error probabilities across all possible configurations, expected queue sizes, and worst-case expected termination times. This is facilitated through the property specification language of PRISM, which incorporates temporal logics like PCTL, CSL, LTL and PCTL*, and includes extensions for quantitative specifications and costs/rewards.

4.1.1 PRISM language

The PRISM language, based on the Reactive Modules formalism of Alur and Henzinger, is a simple, state-based language used to specify the models that PRISM supports for construction and analysis. The language is constructed from fundamental components called modules and variables. A model in PRISM is composed of several modules, each containing local variables. The state of a module is determined by the values of these local variables at a given time, and the global state of the model is the aggregate of the local states of all modules.

The behavior of each module is dictated by a set of commands. A command consists of a guard, which is a predicate over all variables in the model, and a set of updates. If the guard's conditions are met, the module can make a transition, which is specified by each update. Each update not only provides the new values of the module's variables (potentially as a function of other variables) but also assigns a probability or a rate to the corresponding transition. Commands can optionally include an action for annotation or synchronization purposes.

Presented below is an example of a simple Markov Decision Process described using the PRISM language, sourced directly from the PRISM manual [16]. The accompanying explanation of the code, also quoted from the manual, describes the functioning of the system modeled by the code:

Example 4. *„Consider a system comprising two identical processes which must operate under mutual exclusion. Each process can be in one of 3 states: 0,1,2. From state 0, a process will move to state 1 with probability 0.2 and remain in the same state with probability 0.8. From state 1, it tries to move to the critical section: state 2. This can only occur if the other process is not in its critical section. Finally, from state 2, a process will either remain there or move back to state 0 with equal probability."*

```
mdp

module M1

    x : [0..2] init 0;

    [] x=0 -> 0.8:(x'=0) + 0.2:(x'=1);
    [] x=1 & y!=2 -> (x'=2);
    [] x=2 -> 0.5:(x'=2) + 0.5:(x'=0);

endmodule

module M2

    y : [0..2] init 0;

    [] y=0 -> 0.8:(y'=0) + 0.2:(y'=1);
    [] y=1 & x!=2 -> (y'=2);
    [] y=2 -> 0.5:(y'=2) + 0.5:(y'=0);

endmodule
```

In this example, we can see how the concept of mutual exclusion, as well as dynamic behaviours within models supported by PRISM, can be succinctly expressed using the PRISM language.

4.1.2 Reward-based Properties in PRISM

Given that reward mechanisms are central to my work, let's explore the different reward-based properties implemented within the PRISM language.

PRISM allows for models to be supplemented with rewards or costs, providing the capability to analyze properties associated with these expected values. A variety of reward properties can be defined and analyzed, including reachability reward, co-safe LTL reward, cumulative reward, total reward, instantaneous reward, and steady-state reward.

Reachability Rewards: Reachability reward properties pertain to the rewards accumulated along a model path until a specified point is reached. For example, the reward property „F prop" signifies the reward accumulated along a path until a state satisfying the property 'prop' is reached. This type of property is commonly used when model rewards correspond to time, allowing for the expectation of time to reach a certain state to be expressed.

Example 5. *For a simple illustration, consider the following formula:*

```
R{"rew0"}max=? [F "target"]
```

This represents a query for the maximum possible accumulated value in the data variable „rew0" when the agent reaches the state labeled „target".

Cumulative and Total Rewards: Cumulative reward properties denote the reward accumulated along a model path until a given time has passed, as denoted by the property C<=t. Total reward properties, on the other hand, refer to the indefinite accumulation of rewards along a path. Unless a path consistently stays in states with zero rewards, the total reward will be infinite.

For example, a query that computes the expected total cumulative reward over the entire run of model is written as follows:

```
R=? [ C ]
```

Steady-State Rewards: Steady-state reward properties relate to the long-term reward in the model. A common application of this type of property is when model rewards correspond to resources like power consumption.

4.2 STORM

Information about STORM have been sourced from the paper „The probabilistic model checker STORM" by Hensel, Junges, and Katoen [10].

STORM is a sophisticated probabilistic model checker that is designed to tackle the verification of systems that incorporate stochastic uncertainty. This tool, which stands out with its unique features, supports the analysis of both discrete and continuous-time variations of Markov chains and Markov decision processes. STORM accepts multiple input languages for Markov models, including PRISM and JANI modeling languages, dynamic fault trees, generalized stochastic Petri nets, and the probabilistic guarded command language, thereby offering a wide range of applicability.

The tool is distinguished by its modular setup that allows for the easy exchange of solvers and symbolic engines. This flexibility complements its Python API, which enables rapid prototyping by encapsulating STORM's high-speed and scalable algorithms. STORM thus provides a comprehensive toolkit for the quantitative evaluation of system performance alongside correctness, making it an invaluable asset in the realm of probabilistic model checking.

The quantile plot depicted in 4.1 from the study 'The probabilistic model checker STORM' [10], illustrates how STORM outperforms other model checkers, solving more benchmark instances and generally doing so faster.

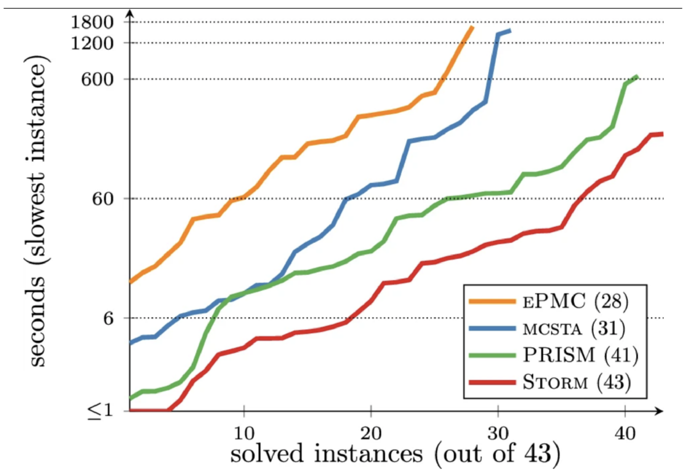

Figure 4.1: Graph depicting the performance of STORM compared to other model checkers, as detailed in the study 'The probabilistic model checker STORM' [10].

4.2.1 Architecture of STORM

STORM's architecture is designed for performance and modularity, with the logical structure of the software divided into various libraries and executables, all dependent on the core 'storm' library. It features two different in-memory representations of Markov models: sparse matrices and Multi-Terminal Binary Decision Diagrams (MTBDDs). Sparse matrices, which use memory roughly proportional to the number of transitions with nonzero probability, are suited for small to medium-sized models, while MTBDDs are better suited for larger models due to their ability to store models compactly.

An interesting aspect of STORM's architecture is the concept of solvers. The tasks related to probabilistic verification often revolve around solving subproblems, and STORM provides abstract interfaces for different solver types that facilitate the solution of these subproblems. It currently includes multiple implementations for each solver interface, further enhancing the tool's flexibility.

STORM is primarily written in C++, with extensive use of template meta-programming. This coding choice allows for high performance through fine-grained control over implementation details like memory allocations and enables type-dependent optimizations at compile time. A large part of the code is written agnostic of the data type (floating point, rational number, or even rational functions), with only the core parts specialized based on the data type.

However, the use of C++ and advanced templating patterns can also present challenges. While it permits easy interfacing with high-performance solvers and data structure libraries, it can be difficult to understand for those unfamiliar with advanced templating patterns and can significantly increase compile times. Despite these challenges, the overall architecture of STORM demonstrates a powerful and flexible approach to probabilistic model checking.

STORM also leverages Python bindings to extend its functionality and accessibility. By employing `pybind11`, a lightweight header-only library that exposes C++ types in Python and vice versa, STORM is able to offer a Python interface. This allows for use-cases where scripting in Python is preferred to utilize the power and performance of STORM's C++ architecture.

4.2.2 POMDP Analysis in STORM

STORM exhibits notable capabilities in the analysis of Partially Observable Markov Decision Processes. It supports three methods for POMDP analysis including (quantitative) reachability verification, policy synthesis under observation-based policies with a fixed memory, and qualitative variant of reachability. These methodologies enable STORM to handle nondeterminism and synthesize policies even in complex, real-world systems where nondeterminism is controllable. These features are particularly important for systems where decision-making is based on incomplete or imperfect observations, further broadening the range of applications STORM can handle.

4.3 PAYNT

This section introduces and discusses PAYNT, an innovative tool that automates the synthesis of finite-state probabilistic programs. Developed by R. Andriushchenko and M. Češka, PAYNT represents a significant leap forward in the sphere of probabilistic program synthesis, using program sketches to describe a finite family of program candidates [4]. At the heart of this tool lies a powerful synergy between inductive oracle-guided methods and advanced probabilistic model checking, enabling PAYNT to reason about all potential program candidates effectively. This section explores the inner workings of PAYNT and its performance in different application domains.

4.3.1 Architecture of PAYNT

PAYNT's architecture 4.2 is built upon the probabilistic model checker, Storm, utilizing a Python API for the synthesis loop's flexible construction, with the high-performance components implemented in C++. The tool accepts either a PRISM or JANI sketch along with a set of temporal properties, and provides a satisfying realization, if available, or reports the absence of such realization [4].

The architecture of PAYNT is organized into multiple modules, including family handlers, chain builders, and model checkers, among others. Family handlers store data about the previously explored design space, employing various methods such as member enumeration and SAT representation. Chain builders generate representations of Markov chains or quotient MDPs based on the provided realizations, which are then verified by the model checkers.

PAYNT also incorporates counterexample generation, using either a MaxSat or a greedy state-expansion approach. It operates through three analysis loops—1-by-1 enumeration, CEGIS, and AR—each with a different approach to the exploration of realizations. The hybrid approach, in particular, combines AR and CEGIS approaches, alternating between the two mid-execution for a comprehensive analysis.

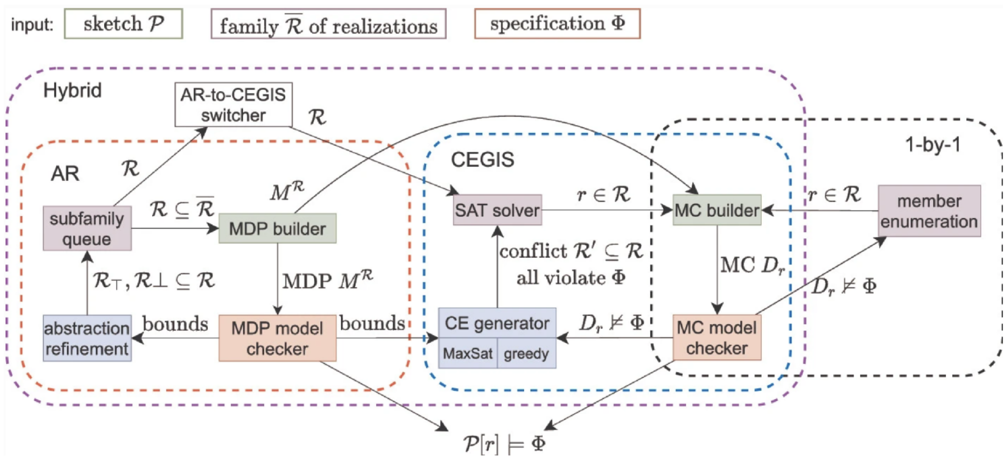

Figure 4.2: Diagram illustrating the architecture of the PAYNT tool, as presented in the paper PAYNT: A Tool for Inductive Synthesis of Probabilistic Programs [4].

4.3.2 Efficient Synthesis Using PAYNT

The utilization of PAYNT can be demonstrated through a synthesis problem originally listed in PAYNT article [4]. This problem involves a request-processing server with a queue capacity of Q_{max}, captured in image 4.3. When the queue is full, incoming requests are discarded. The server has three power states – sleeping, idle, and active, with the latter being the only one that processes requests. Transitioning from lower-energy states to active requires additional energy and time. The server's power consumption during processing depends on the queue size, and the server's operational time is random but limited.

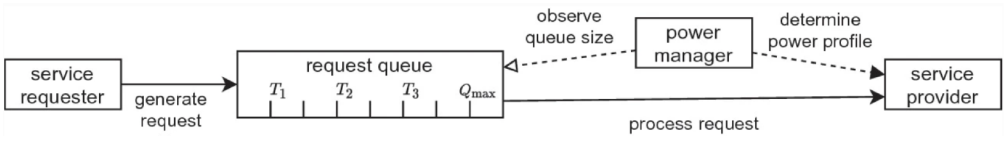

Figure 4.3: The server for request processing [4].

Control over the server's power state is in the hands of a power manager (PM), which determines the power state based on the current queue size, divided into four occupancy levels set by thresholds T1, T2, and T3. Each threshold is a parameter representing a fraction of the queue capacity, and each is associated with a power profile P1 to P4, corresponding to the three power states.

PAYNT operates on a sketch – a PRISM or JANI language program with undefined parameters (holes). Each module in the program has variables and state transitions, expressed as guarded commands. When a guard condition is met, variables are updated based on a probabilistic distribution. The sketch inputs include the queue capacity ($Q\_max \in \{1, \ldots, 10\}$), thresholds ($T1 \in \{0, 0.1, 0.2, 0.3, 0.4\}$, $T2 \in \{0.5\}$, $T3 \in \{0.6, 0.7, 0.8, 0.9\}$), and corresponding power profiles (P1 to P4 $\in \{0, 1, 2\}$), leading to a design space of 16,200 different power managers.

PAYNT's objective is to instantiate the holes such that power consumption is minimized and the expected number of lost requests is less than 1. Given a sketch and a specification, PAYNT searches the design space for a hole assignment that satisfies the specification, or reports that none exists. For example, PAYNT found an optimal power manager configuration in just one minute, three times faster than naive enumeration. Furthermore, in a complex problem with about 43M solutions, PAYNT found the optimal power manager within 10 hours, compared to over a month using enumeration [4].

4.3.3 Algorithmic overview of PAYNT

Presented below is an overview of the procedural steps involved in the PAYNT algorithm.

Algorithm 6 PAYNT

```
 1: procedure PAYNT(sketch, properties)
 2:     Load the sketch and properties.
 3:     Initialize the appropriate synthesizer.
 4:     Synthesize an assignment for the design space of the problem.
 5:     If a valid assignment is synthesized, build a Markov chain from it.
 6:     Check the satisfiability of the assignment against the problem's specification.
 7:     if Synthesis was able to find a solution then
 8:         Use the synthesized assignment and its satisfiability.
 9:     else
10:         Split the problem into smaller subfamilies
11:         Repeat the process from step 6 for each.
12:     end if
13:     for each sub-problem do
14:         Verify feasibility of sub-problem.
15:         Analyze each sub-problem to determine if it can be improved
16:         if a sub-problem can be improved then
17:             Find an improving assignment
18:         else
19:             Explore other branches of the problem space.
20:         end if
21:     end for
22: end procedure
```

The PAYNT algorithm proceeds through several steps to solve a given problem using sketch and properties. Firstly, the sketch and properties are loaded into the system, which represents the initial problem. Then, an appropriate synthesizer is initialized to generate potential solutions for the problem.

The synthesizer begins by trying to synthesize an assignment for the design space of the problem. If a valid assignment is synthesized, the algorithm builds a Markov chain from this assignment. It then checks the satisfiability of the assignment against the problem's specification.

If the synthesized assignment is a solution to the problem, the algorithm uses the synthesized assignment and its satisfiability.

However, if the synthesizer does not find a solution in the initial attempt, it splits the problem into smaller subfamilies. It then repeats the process of synthesizing an assignment and checking its satisfiability for each subfamily.

For each sub-problem (subfamily), the algorithm verifies the feasibility of the sub-problem and analyzes it to determine if it can be improved. If a sub-problem can be improved, the algorithm finds an improving assignment for it. If it cannot be improved, the algorithm explores other branches of the problem space.

This procedure continues until either a satisfactory solution is found or all branches of the problem space have been explored.

4.4 Conclusion

In conclusion, PRISM, STORM, and PAYNT each bring unique and powerful capabilities to the table in the field of probabilistic model checking and program synthesis. PRISM's robust language and model checking capabilities provide a solid foundation, handling a wide variety of probabilistic models. STORM takes this further, offering a highly performant and extensible framework that deals with larger models and complex properties with great efficiency. PAYNT takes advantage of both worlds. It not only leverages the PRISM language for its inductive synthesis technique of probabilistic programs but also utilizes STORM's computational power for efficient exploration of design spaces and delivery of optimal solutions. Each of these tools has its strengths, and together, they comprise a comprehensive toolkit for addressing a wide range of challenges in probabilistic modeling and synthesis. Their combined capabilities pave the way for exciting future advancements in this field [14, 10, 4].

Chapter 5

Implementation and Experimental Evaluation

In this chapter, I aim to comprehensively articulate contributions to the development of the PAYNT tool made throughout this work. This exploration will include a experimental evaluation of the implemented enhancements. The intention is to discuss the impact of these changes and their role in advancing the tool's capabilities and performance.

5.1 Discount Factor Transformation

The primary objective of this thesis is to address an identified limitation within the PAYNT tool, specifically, the issue of discount factor transformation. This section of the present chapter is devoted to elucidating this concept and detailing the specific concerns associated with it in the context of PAYNT.

In the context of Markov decision processes (MDPs), the discount factor γ plays a crucial role in determining the trade-off between immediate and future rewards. To transform a discounted MDP into an undiscounted one, a sink state was introduced. A sink state is an absorbing state where the process remains once it is entered, with no possibility of leaving. In the transformed MDP, we add transitions to this sink state for every existing state, with a probability of $(1-\gamma)$. Simultaneously, we scale the rewards of all transitions by a factor of γ. Consequently, the transformed MDP now operates in an undiscounted setting ($\gamma = 1$), while preserving the essence of the original discounted problem. By doing so, we effectively capture the essence of the discount factor through the sink state, enabling the utilization of undiscounted MDP algorithms for solving the original problem.

In following example, discount factor transformation is displayed on specific values.

Example 6. *Let's consider simple Markov chain with three states, as depicted on 5.1 with discount $\gamma = 0.9$.*

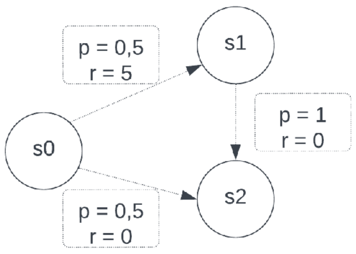

Figure 5.1: Simple Markov chain before application of discount factor transformation

To apply discount factor transformation, we add new state („sink state“) and transitions from each of previous states into sink states with transition probability of $1 - \gamma = 0.1$*. All rewards are multiplied by* γ *and all previous transition probabilities are also multiplied by* γ*. Markov chain now fully operates in an undiscounted setting. Result is shown in 5.2.*

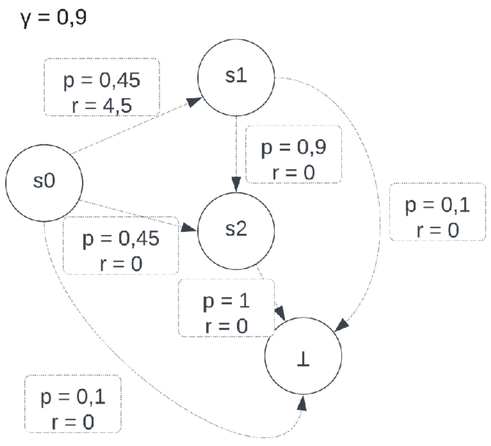

Figure 5.2: Simple Markov chain after applying discount factor transformation

However, it is important to note that this transformation inevitably increases the size of the model. The addition of new transitions to the sink state from all existing states results in a more complex model, which can present challenges when processing or analyzing the MDP. Larger models are inherently more difficult to work with due to their increased computational demands and the greater potential for complexity in their structure and dynamics.

5.2 Discounted Value Iteration

The primary objective of this research was to address a key limitation of PAYNT concerning the handling of discounted properties. This was accomplished by extending the functionality of the STORM tool to incorporate the discounted variant of the value iteration method. This task was fraught with multiple challenges. Specifically, it necessitated the expansion of data classes within PAYNT and the addition of Python bindings to STORM, thereby ensuring an interface between the Python scripts and the STORM library. Furthermore, it also required the implementation of abstract template methods, which was essential to ensure conformity with the existing structure of STORM.

5.2.1 Discount Factor

In the original PAYNT pipeline, the model was parsed and transformed prior to being checked, resulting in the discount factor becoming redundant during the subsequent value iteration stage. To ensure the accessibility of the discount factor throughout the entire process of model checking, I introduced a series of extensions and modifications.

The first step was to augment the `MinMaxSolverEnvironment` class in PAYNT with an additional attribute named `discount`. This attribute required corresponding getter and setter methods, which were defined within the `MinMaxSolverEnvironment` in STORM. Finally, I established a connection between STORM and the Python scripts by creating a binding in STORMPY.

Through these modifications, the discount factor was made readily available during all stages of the model checking process, significantly enhancing the versatility and effectiveness of the PAYNT tool.

5.2.2 Value Iteration Enhancement

With the discount factor now accessible throughout the model checking process, I was able to proceed with the development of the discounted value iteration function. A preliminary step was to identify the appropriate solver environment, given the multiple minmax solver environments implemented within STORM. The `IterativeMinMaxLinearEquationSolver` was established as the environment in use.

My next task was to identify a section of the code where the program could discern whether or not the discount factor was being utilized. If the discount was not in use, the process should continue with the undiscounted value iteration. Conversely, if the discount was in play, the discounted version of value iteration should be used.

I pinpointed this critical juncture in the `solveEquationsValueIteration` function, which controls the execution of the value iteration. I introduced a decision-making condition that checks whether the discount factor differs from 1. If this is the case, the discounted value iteration is initiated by calling the newly added `performValueIterationDiscounted` method. This effectively integrates the use of discounted value iteration within the existing framework of STORM.

Delving further into the code, I discovered that Gauss-Seidel multiplication was employed to enhance STORM's computational capabilities. This operation ultimately triggers the `MultiplyAndReduceBackwardDiscounted` method, a template function which implements pointer arithmetic and vector multiplication - foundational elements of value iteration.

The discount factor is passed down to this level, ensuring its inclusion in the most critical computational step. By integrating the discount factor into the heart of the computation process, I ensured its impact was felt throughout the entirety of the value iteration, thereby advancing the overall capacity of the PAYNT tool in handling discounted properties.

5.2.3 Limitations of STORM

Upon implementing the discounted value iteration, my next step was to attempt to bypass the discount factor transformation step, relying solely on the newly integrated discounted value iteration. However, I encountered a challenge. For PAYNT to effectively check the model on which value iteration is applied, it requires a PRISM formula. This formula, parsed within STORM, serves as a determinant for the properties that are being checked. Therefore, the elimination of the discount factor transformation step was not feasible, as it would compromise the integrity of the model checking process.

With the optimality objective defined as `R[exp]„reward"max=?[F „discount_sink"]`, STORM essentially seeks the maximum expected reward that can be accumulated, given optimal action choices, before reaching a state labeled 'discount_sink'. After the removal of the discount factor transformation, this 'discount_sink' state was no longer present in the model. Consequently, I attempted to utilize other formulas, such as those concerning steady state or total cumulative reward, which query about rewards over an infinite time horizon. However, I discovered that these formulas were not yet implemented within the STORM framework.

5.2.4 Evaluation of Discounted Value Iteration

Incorporating additional optimality objectives into STORM extended beyond scope of this project, presenting a significant challenge. To bypass this issue and test the functionality of my implementation, I chose to employ the discount factor transformation once more, albeit with a discount factor approaching one (0.99999...) to mimic as closely as possible the behavior of the untransformed model. This step, however, meant sacrificing the benefits of model size efficiency for the chance to validate the outcomes of the discounted value iteration. The value iteration process generates vectors of potential optima at each iteration, from which the appropriate optimum is later selected by PAYNT. As such, I decided to compare these vectors between the discounted and undiscounted value iterations.

5.3 Experimental Evaluation

In the Experimental Evaluation section, we conduct an in-depth analysis of various models, focusing on the model's optima, the error associated with the Discounted Value Iteration (Discounted VI) in comparison to the Value Iteration with Discount Factor Transformation (VI + DTF), and the time difference required for model checking. These evaluations were performed using models available in the PAYNT's GitHub repository[1].

A summary of these experiments is presented in Table 5.1. This table provides a sample of the models tested, detailing the optimum value, the absolute error by which Discounted VI differs from VI + DTF, and the respective original time and improvement with Discounted VI for each model.

It is critical to highlight that the results of the Discounted VI synthesis are skewed due to the presence of the discount factor transformation, a necessity previously discussed in Section 5.2.4.

[1]Accessible at https://github.com/randriu/synthesis

Model Name	**Model Optimum**	**Error**	**Original Time [s]**	**Improvement [s]**
1d.noisy	1.0693	0.0000	1.6763	0.2179
4x5x2.95	0.2323	0.0142	1.7600	0.2202
shuttle.95	18.8960	0.0000	1.7025	-0.1149
cheng.D5-1	64899.0102	5447.7890	20.2912	-0.2396
stand-tiger.95	0.0	69.2908	2.0791	-1.1588
ejs5	1.0	0.0	1.7848	0.2889
cheng.D3-4	61510.4417	9449.3947	5.0684	1.0694
web-mall	0.4005	0.0	1.8546	-0.0245
cheese.95	0.6523	0.1025	1.7507	-0.0209
network	82.9761	62.3134	1.8036	-0.1218
ejs6	1.0	0.0	2.0416	0.2015
4x4.95	0.5167	0.0383	1.9749	0.0701
learning.c2	1.0182	0.4111	8.4710	3.9977
1d	0.9536	0.0	2.045	0.1575
concert	0.0	0.0	1.9391	0.0717
ejs7	1.0	0.0007	1.7914	-0.1205
ejs2	15156.5455	0.0159	2.3354	-0.0920
ejs1	25003.3153	0.0	2.1903	0.1207
cheng.D3-5	117600.8146	0.0714	2.8518	0.1853
line4-2goals	0.4658	0.0	1.9216	0.0575
milos-aaai97	29.4586	1.0924	49.0348	46.9239
mini-hall2	2.5581	0.9625	1.9629	0.0254
cheng.D3-1	74656.3104	5025.017	5.9109	1.7440
4x3.95	0.4418	0.6189	2.1347	-0.9012

Table 5.1: Results for selected models.

To facilitate comprehension of the table, we provide some statistical measurements: The median error of the Discounted VI is 4.92 %, while the mean time improvement of the Discounted VI over the VI + DTF is 2.1067 s, with a median time improvement of 0.0962 s. On average, the Discounted VI took 3.297 s for model checking, while the VI + DTF took 5.4037 s.

The results suggest that the Discounted Value Iteration (Discounted VI) method delivers results of sufficient precision. However, to determine its absolute precision, we would need to test the Discounted VI without the involvement of discount factor transformation, a process which is not currently feasible. Moreover, it can be deduced from the results that Discounted VI doesn't hinder the model checking process. On the contrary, it appears to enhance the speed of the process, albeit marginally.

Chapter 6

Final Considerations

In this thesis, my primary objective was to optimize the synthesis of POMDP controllers with discounted reward properties. My initial efforts were focused on understanding the current state-of-the-art methods in model checking, with particular emphasis on abstraction refinement and inductive synthesis. Subsequently, I delved into the realm of existing probabilistic model checking tools to gain a comprehensive understanding of their functionalities and limitations. In the next phase, I proposed an optimization strategy for the PAYNT tool by incorporating a discounted value iteration method within the STORM tool. Upon successful implementation, a modest increase in the speed of the model checking process was observed, indicating that this approach surpasses the previous one in terms of efficiency. It can be inferred that eliminating the discount factor transformation step would further enhance this process.

Future enhancements to the approach I've proposed are both necessary and promising. Within the context of STORM, there is potential for the implementation of additional optimality objectives, which could broaden the tool's applicability. As for PAYNT, a significant advancement could be achieved by successfully eliminating the need for discount factor transformation, which would streamline the process and further optimize the synthesis of POMDP controllers.

Bibliography

[1] Amato, C., Bonet, B. and Zilberstein, S. Finite-state controllers based on Mealy machines for centralized and decentralized POMDPs. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. 2010, vol. 24, no. 1, p. 1052–1058.

[2] Andriushchenko, R., Češka, M., Junges, S. and Katoen, J. Inductive synthesis of finite-state controllers for POMDPs. In: *UAI'22. Proceedings of Machine Learning Research*.

[3] Andriushchenko, R. *Computer-Aided Synthesis of Probabilistic Models*. Brno, 2020. Master's Thesis. Brno University of Technology, Faculty of Information Technology. Supervisor RNDr. Milan Češka, P.

[4] Andriushchenko, R., Češka, M., Junges, S., Katoen, J.-P. and Stupinský, PAYNT: A Tool for Inductive Synthesis of Probabilistic Programs. In: Silva, A. and Leino, K. R. M., ed. *Computer Aided Verification*. Cham: Springer International Publishing, 2021, p. 856–869. ISBN 978-3-030-81685-8.

[5] Braziunas, D. POMDP solution methods. *University of Toronto*. 2003.

[6] Češka, M., Hensel, C., Junges, S. and Katoen, J.-P. Counterexample-driven synthesis for probabilistic program sketches. In: Springer. *Formal Methods–The Next 30 Years: Third World Congress, FM 2019, Porto, Portugal, October 7–11, 2019, Proceedings*. 2019, p. 101–120.

[7] Češka, M., Hensel, C., Junges, S. and Katoen, J.-P. Counterexample-guided inductive synthesis for probabilistic systems. *Formal Aspects of Computing*. Springer. 2021, vol. 33, 4-5, p. 637–667.

[8] Češka, M., Jansen, N., Junges, S. and Katoen, J.-P. Shepherding hordes of Markov chains. In: Springer. *Tools and Algorithms for the Construction and Analysis of Systems: 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings, Part II 25*. 2019, p. 172–190.

[9] Grzes, M., Poupart, P. and Hoey, J. Isomorph-free branch and bound search for finite state controllers. 2013.

[10] Hensel, C., Junges, S., Katoen, J.-P., Quatmann, T. and Volk, M. The probabilistic model checker Storm. *International Journal on Software Tools for Technology Transfer*. Springer. 2021, p. 1–22.

[11] KNUTH, D. and YAO, A. Algorithms and Complexity: New Directions and Recent Results. In:. Academic Press, 1976, chap. The complexity of nonuniform random number generation.

[12] KOCHENDERFER, M. J. *Algorithms for Decision Making*. Cambridge, MA: MIT Press, 2022.

[13] KURNIAWATI, H., HSU, D. and LEE, W. S. Sarsop: Efficient point-based pomdp planning by approximating optimally reachable belief spaces. In: Citeseer. *Robotics: Science and systems*. 2008, vol. 2008.

[14] KWIATKOWSKA, M., NORMAN, G. and PARKER, D. PRISM 4.0: Verification of Probabilistic Real-time Systems. In: GOPALAKRISHNAN, G. and QADEER, S., ed. *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*. Springer, 2011, vol. 6806, p. 585–591. LNCS.

[15] KWIATKOWSKA, M., NORMAN, G. and PARKER, D. Stochastic model checking. *Formal Methods for Performance Evaluation: 7th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2007, Bertinoro, Italy, May 28-June 2, 2007, Advanced Lectures 7*. Springer. 2007, p. 220–270.

[16] PRISM PROJECT TEAM. *PRISM - PRobabilistic Symbolic Model Checker*. 2023. Accessed: 30-April-2023. Available at: https://www.prismmodelchecker.org.