BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

CONTROLLING AUTONOMOUS SYSTEMS BASED ON PARTIALLY OBSERVABLE MARKOV DECISION PROCESSES

ŘÍZENÍ AUTONOMNÍCH SYSTÉMŮ ZALOŽENÉ NA MARKOVSKÝCH MODELECH S ČÁSTEČNÝM POZOROVÁNÍM

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTOR **JULIE GYSELOVÁ**
AUTOR PRÁCE

SUPERVISOR **doc. RNDr. MILAN ČEŠKA, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2022

Department of Intelligent Systems (DITS) Academic year 2021/2022

Bachelor's Thesis Specification

24626

Student:	**Gyselová Julie**
Programme:	Information Technology
Title:	**Controlling Autonomous Systems Based on Partially Observable Markov Decision Processes**
Category:	Formal Verification

Assignment:

1. Study the existing methods for controlling partially observable Markov decision processes. Focus on methods using finite-state controllers.
2. Evaluate these method in the context of controlling autonomous systems.
3. Design improvements and extensions of these methods allowing efficient optimal synthesis for various classes of the controllers.
4. Implement these improvements and extensions on top of the tool PAYNT for synthesis of probabilistic programs.
5. Perform a detailed experimental evaluation of the proposed methods on a suitable benchmark.

Recommended literature:

- Kochenderfer, M.J., Wheeler, T.A., and Wray K.H, Algorithms for Decision Making, MIT Press 2022.
- Andriushchenko, R., Češka, M., Junges, S., Katoen, J.P. and Stupinský, Š. PAYNT: A Tool for Inductive Synthesis of Probabilistic Programs. In *CAV 2021*.
- Junges, S., N. Jansen, R. Wimmer, T. Quatmann, L. Winterer, J. P. Katoen, and B. Becker. Finite-state controllers of POMDPs using parameter synthesis. In *UAI 2018.*
- Kumar, A. and Zilberstein, S. History-based controller design and optimization for partially observable MDPs. In *ICAPS 2015*.
- Wray, K.H. and Czuprynski, K., Scalable POMDP Decision-Making Using Circulant Controllers.In *ICRA 2021.*

Requirements for the first semester:

- Items 1, 2 and partially item 3.

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

Supervisor:	**Češka Milan, doc. RNDr., Ph.D.**
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	November 1, 2021
Submission deadline:	July 29, 2022
Approval date:	November 3, 2021

Abstract

Partially observable Markov decision processes offer a way to model systems with state uncertainty. An agent has limited information (observation) about its current location in the system. A finite-state controller that translates this information to actions that the agent can perform helps the agent interact with the model and achieve its goals. PAYNT is a tool that constructs a design space that contains all possible finite-state controllers of a given size for a POMDP and then tries to find the best FSC among those. In this thesis, I introduce a way to restrict the design space to encode only a subset of the controllers so that PAYNT can find the best controller in a much shorter time. If the used restriction is suitable, the controller quality is not affected. I also implement a method that can make the synthesis method implemented in PAYNT continuously find FSCs of increasing sizes and improving qualities by gradually applying restrictions from a predefined set.

Abstrakt

Systémy se stavovou neurčitostí lze modelovat pomocí Markovských rozhodovacích procesů s částečným pozorováním. Agent, který se v takovém systému pohybuje, má o své pozici v rámci systému pouze omezené informace (pozorování). Konečně-stavový kontroler umí přiřadit vhodnou akci k aktuálnímu pozorování. Díky tomu může agent se systémem lépe interagovat a dobrat se svého cíle. Nástroj PAYNT umí najít nejkvalitnější kontroler mezi všemi možnými kontrolery dané velikosti pro daný model. V této práci představím způsob, jakým lze omezit designový prostor, ve kterém PAYNT kontrolery hledá, tak, aby zakódovával pouze určitou podmnožinu kontrolerů, která lze vyhodnotit v menším čase. Pokud je použita vhodná restrikce, kvalita kontrolerů není ovlivněna. Dále implementuji metodu, která postupně aplikuje tyto restrikce na designový prostor a umožňuje syntetizační metodě v PAYNTu nepřetržitě hledat kontrolery větších velikostí a lepší kvality.

Keywords

Partially observable Markov decision processes, finite-state controller synthesis

Klíčová slova

Markovské rozhodovací procesy s částečným pozorováním, syntéza konečně-stavových kontrolerů

Reference

GYSELOVÁ, Julie. *Controlling Autonomous Systems Based on Partially Observable Markov Decision Processes*. Brno, 2022. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. RNDr. Milan Češka, Ph.D.

Controlling Autonomous Systems Based on Partially Observable Markov Decision Processes

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of doc. RNDr. Milan Češka, Ph.D. Supplementary information was provided by Ing. Roman Andriushchenko. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Julie Gyselová
July 29, 2022

Contents

Chapter 1

Introduction

Creating autonomous systems that could operate without human input has been a long-term goal of mankind. These systems can range from simple machines designed to perform a simple task over and over again to self-driving vehicles that are yet to drive on our roads. For the latter, the biggest obstacle is the surrounding environment. What is natural to navigate for humans is not as easy for a man-made machine.

The objective of this thesis is not going to be creating a driver-less car. Examples here will be much simpler than that – navigating grids and two-dimensional mazes. The challenge, however, will be similar to that of self-driving cars. How to solve a problem when we only have limited knowledge about the surroundings?

"*Outcome uncertainty*, where the effects of our actions are uncertain, *model uncertainty*, where our model of the problem is uncertain, *state uncertainty*, where the true state of the environment is uncertain, and *interaction uncertainty*, where the behavior of the other agents interacting in the environment is uncertain." [11]

This thesis will focus on systems with state uncertainty and how it can be modeler using Partially observable Markov decision processes that define the system's states, the limited information which describes these states, available actions and the results of these actions.

There are two approaches to controlling POMDPs, or rather controlling agents interacting with POMDPs, the first being *belief based* – where a belief distribution is kept and updated based on taken actions and received observations and these beliefs are then mapped to actions for the agent to take [11] – and the second one is using *finite-state controllers* (FSCs). Finite-state controllers map observations to actions either just based on this information alone or by using their own inner state to provide additional information about the history of interactions between the agent and the system [12].

An optimal controller can be synthesized from a design space that encompasses all possible controllers of a set size using the Python tool PAYNT (Probabilistic progrAm sYNThesizer) [1, 3]. The design space size, however, grows exponentially with the controller's memory size.

Inspired by Scalable POMDP Decision-Making Using Circulant Controlle by Wray et. al. [14] which focused on synthesizing circulant controllers for agents with cyclic behavioural patterns, I set out to find a way to synthesize only controllers of a certain type using PAYNT while also cutting back on synthesis time of the growing design spaces.

To achieve this, I introduce *restrictions.* A suitable restriction offers a way to only synthesize controllers of a certain type while reducing the synthesis time significantly without affecting the controller quality. This thesis explores different restrictions and compares the resulting controllers to those synthesized from unrestricted design spaces.

Taking a set of different, gradually less strict restrictions and sequentially applying it to the growing design spaces should allow PAYNT to find controllers of increasing sizes and improving qualities. The results of this, *incremental*, approach are compared to the results of iterative synthesis from non-restricted design spaces and to the memory injection strategy by Andriushchenko et. al. introduced in Inductive Synthesis of Finite-State Controllers for POMDPs [1].

The preliminary experiments using the introduced restrictions show promising results in three of the four benchmarks used.

Chapter 2

Preliminaries

In this chapter I explain the most essential stochastic, probability driven models in this thesis and using easy to understand examples I explain how they relate to controlling autonomous systems.

Definitions in this chapter are taken over and adapted from Stochastic Model Checking [13] (definition 1), Shepherding Hordes of Markov Chains [15] (definitions 2 and 3), and from Finite-state Controllers of POMDPs via Parameter Synthesis [9] (definitions 4 and 5).

2.1 Markov Chains

Markov chains (MCs) are the most elementary kind of Markov models and they are essential to comprehending other models that are derived from them. A stochastic model is considered *markovian* if it holds the Markov property – the next state is determined by the current state only.

Markov chains can be divided into two categories – *discrete-time Markov chains* (DTMCs) which operate on discrete time points and *continuous-time Markov chains* (CTMCs) which operate on time intervals. For the purposes of this thesis, Markov chains will refer to DTMCs.

Definition 1. A *discrete-time Markov Chain* M is defined by a tuple $M = (S, s_0, P)$. S is a finite set of states, s_0 is the initial state, and $P : S \times S \to [0, 1]$ is the *transition probability matrix* where $\sum_{s \in S} P(s, s') = 1$ for all $s \in S$ which denotes the probability that a MC in state s can transition into state s' in one step.

If a state only has a single possible transition and that is back to itself, it is called an *absorbing state*. In such a state, $P(s, s) = 1$ and $P(s, t) = 0$ for all states $t \neq s$ [4]. Absorbing states can be viewed as end states, once they are reached, no other states can be transitioned to.

A *path* ω is a non-empty sequence of states $s_0 s_1 s_2 ...$ that can be either finite or infinite where $i \in \mathbb{N}_0, s_i \in S$ and $P(s_i, s_{i+1}) > 0$. The length of this sequence (the number of transitions) is the path's length [13].

A substantial part of working with Markov chains and models derived from MCs revolves around *reachability*, the property that describes whether it is possible to transition from state s_0 to state s_n, with what probability, and in how many of transitions. A state s_n is reachable if there exists a path from s_0 to s_n.

Example 2.1.1. A common example used with Markov chains is the Knuth-Yao dice, a model that sets out to simulate an N-sided dice using a coin flip [10]. What would the Markov chain for simulating a three sided dice with a fair coin look like?

Solution. One coin-flip can differentiate between the end states **1** and **2** starting from an intermediate state s_1. Tossing tails in the intermediate state s_2 determines a transition to absorbing state **3**. Because the number three is not a power of two, we need a way to redistribute the remaining heads toss into all of the three possible end states. That is done by adding a transition from s_2 back to s_0.

A graphical representation of this MC where red edges (pointing to s_0, s_1, and **1**) represent heads, blue edges (pointing to s_2, **2**, and **3**) represent tails can be seen in figure 2.1.

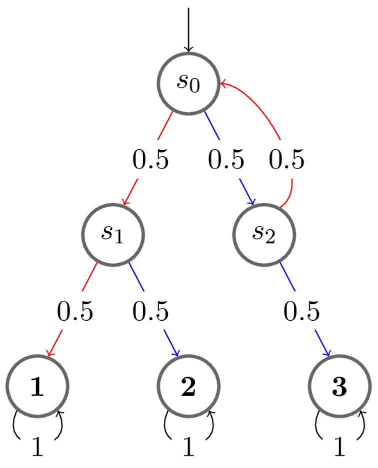


Figure 2.1: Markov chain for simulating a three-sided dice with a coin flip.

The MC is formally defined by the set of states $S = \{s_0, s_1, s_2, \mathbf{1}, \mathbf{2}, \mathbf{3}\}$ where s_0 is the initial state and a transition matrix $P = \begin{pmatrix} 0 & 0.5 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.5 & 0.5 & 0 \\ 0.5 & 0 & 0 & 0 & 0 & 0.5 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$.

The correctness of this MC can be verified by computing the *reachability probabilities* of the states, that is the probabilities of paths of certain lengths ending in those states [4]. Table 2.1 shows the reachability probabilities for all states in the first 10 steps. By step 6, the the reachability probabilities of states **1**, **2**, and **3** are equal to ≈ 0.33 which is the expected value for a fair three-sided dice.

The remaining probability values that are "stored" in states s_0 or states s_1 and s_2 will continually keep getting redistributed to states **1**, **2**, and **3** in subsequent steps.

An example of a path in this MC would be $\omega = \{s_0, s_2, s_0, s_1, \mathbf{1}\}$ and that would be the result of tails once and then heads three times in a row.

State/Step	0	1	2	3	4	5	6	7	8	9	10
s_0	1	0	0.25	0	0.063	0	0.016	0	0.004	0	0.001
s_1	0	0.5	0	0.125	0	0.031	0	0.008	0	0.002	0
s_2	0	0.5	0	0.125	0	0.031	0	0.008	0	0.002	0
1	0	0	0.25	0.250	0.313	0.313	0.328	0.328	0.332	0.332	0.333
2	0	0	0.25	0.250	0.313	0.313	0.328	0.328	0.332	0.332	0.333
3	0	0	0.25	0.250	0.313	0.313	0.328	0.328	0.332	0.332	0.333

Table 2.1: Table reachability probabilities in each state in the span of 10 steps.

2.1.1 Families of Markov Chains

With parameters, a family of Markov chains can compactly describe many different *realisations*, Markov chains defined over the same set of states but with different topologies. While this is interesting on its own, families of Markov chains become most relevant when discussing finite-state controllers and their synthesis.

Definition 2. A family of Markov chains is defined as a tuple $\mathfrak{D} = (S, s_0, K, \mathfrak{P})$ where S is a finite set of states, $s_0 \in S$ is the initial state, K is a finite set of parameters such that the domain of each parameter $k \in K$ is $T_k \subseteq S$, and $\mathfrak{P} : S \rightarrow Distr(K)$ is a family of transition probability matrices.

Definition 3. A realisation of a family of Markov chains $\mathfrak{D}$ is a function $r : K \rightarrow S$ where $\forall k \in K : r(k) \in T_k$. A realisation r yields a MC $D_r = (S, s_0, \mathfrak{P}(r))$, where $\mathfrak{P}(r)$ is the transition probability matrix in which each $k \in K$ in $\mathfrak{P}$ is replaced by $r(k)$. Let $R^{\mathfrak{D}}$ denote the set of all realisations for $\mathfrak{D}$.

Markov chains that belong to the same family have the same set of states, the same initial state, but different transition matrices and topologies. Different realisations of one family can have different reachable states.

Example 2.1.2. Figure 2.2 depicts a family of MCs $\mathfrak{D}$ defined by the set of states $S = \{s_0, s_1, s_2\}$, initial state s_0, set of parameters $K = \{a, b\}$, parameter domains $T_a = \{s_0, s_1\}$ and $T_b = \{s_1, s_2\}$, and transition probability matrix $\mathfrak{P}(0) = 1/2s_1 + 1/2a$, $\mathfrak{P}(1) = 1/2a + 1/2b$, and $\mathfrak{P}(2) = 1/2s_1 + 1/2b$. (Transition probabilities are omitted for the sake of readability.)

What would two possible realisations of this family look like?

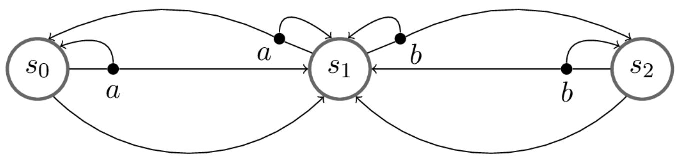


Figure 2.2: Family of Markov chains $\mathfrak{D}$.

Solution. Realisation r_1: $r_1(a) = s_0$, $r_1(b) = s_2$ is depicted in figure 2.3. All states reachable in this realisation, no state is absorbing.

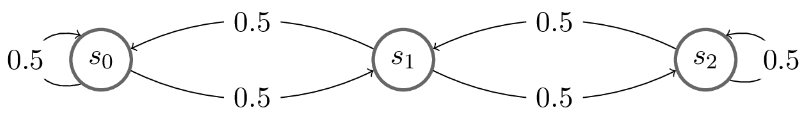


Figure 2.3: Graphical representation of realisation r_1 of the family of MCs $\mathfrak{D}$.

Realisation r_2: $r_2(a) = s_1$, $r_2(b) = s_1$ is depicted in figure 2.4. State s_2 unreachable from the initial state, s_1 is an absorbing state which is always reached after the first step.

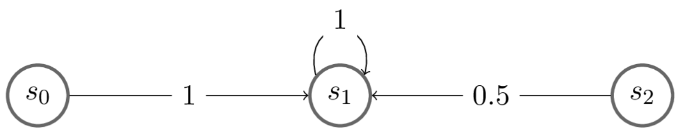


Figure 2.4: Graphical representation of realisation r_2 of the family of MCs $\mathfrak{D}$.

2.2 Markov Decision Processes

Markov Decision Processes (MDPs) are the first extension over Markov chains. MDPs add an *action* between the state and the available transitions. Actions are performed by an *agent*, a real or imaginary entity that can interact with the modeled system. The outcome of each action is given by a probability distribution over a subset of states.

Definition 4. Markov decision processes are a tuple $M = (S, s_0, Act, \mathcal{P})$ where S is a finite set of states, $s_0 \in S$ is the initial state, Act is a finite set of actions, and $\mathcal{P} : S \times Act \times S \rightarrow [0, 1]$ is its transition function where $\sum_{s' \in S} \mathcal{P}(s, a, s') = 1$ applies to all $a \in Act$ and $s, s' \in S$ if action a is available in s (otherwise the sum is equal to 0).

In Markov decision processes, we differentiate between two types of choices – *probabilistic*, and *non-deterministic*. Probabilistic choices are the outcomes of the executed actions and they are determined by the transition function and, if the result of the transition function is a distribution over more than one state, a random number generator. Non-deterministic choices are the actions $a \in Act$. These are decided by an outside entity, e.g. a *controller*, which is sometimes also called a *scheduler* or *policy*.

There are several ways a controller could decide which action should the agent take. Either the controller is *deterministic* and the sequence of actions will always be the same, or it can be *randomised* and the sequence of actions can differ between runs.

Once the action is decided, the MDP behaves just like a simple Markov chain. If the actions are known before the run, a MDP can be transformed to a MC.

Example 2.2.1. The MDP M will represent a baby with two states – *hungry*, which is the initial state, and *sated* – and two actions – *feed* and *ignore*. When a caretaker feeds a hungry baby there is a 20% chance that the baby will stay hungry and an 80% chance that it gets sated, if they ignore the hungry baby, it stays hungry. If the caretaker feeds a sated baby, it will stay sated. Ignoring a sated baby has a 20% chance that the baby gets hungry. [1]

Solution. In each state an agent, the caretaker, is able to perform one of the two actions – *feed* the baby or *ignore* the baby. Depending on the current state of the MDP and the

[1]This example was inspired by a problem described in the book Algorithms for Decision Making [11].

caretaker's action (the deterministic choice) and a random chance (the non-deterministic choice) the process either changes it's state from *hungry* to *sated*, the other way around, or it stays in its current state. The MDP M is depicted in figure 2.5.

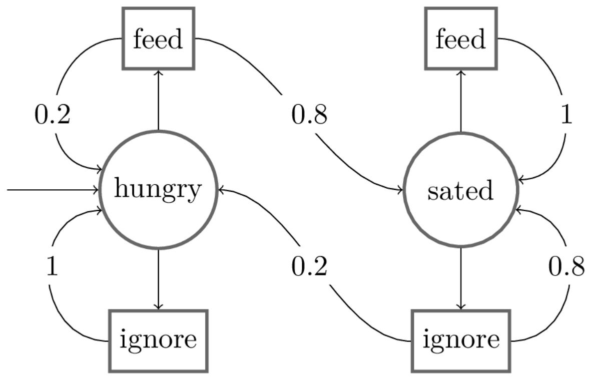


Figure 2.5: The MDP M representing the baby in example 2.2.1

Choosing actions reduces the MDP to a plain Markov chain. Figure 2.6 depicts a MC derived from the MDP in Example 2.2.1 given that the agent chooses to alternate between the actions *feed* and *ignore*. Blue edges (pointing to states $hungry_0$ and $sated_1$) represent transitions after ignoring the baby, red (pointing to states $sated_0$, $hungry_1$, and $sated_2$) after feeding it. Note that there is no path in this MC where two edges of the same color could be taken consecutively.

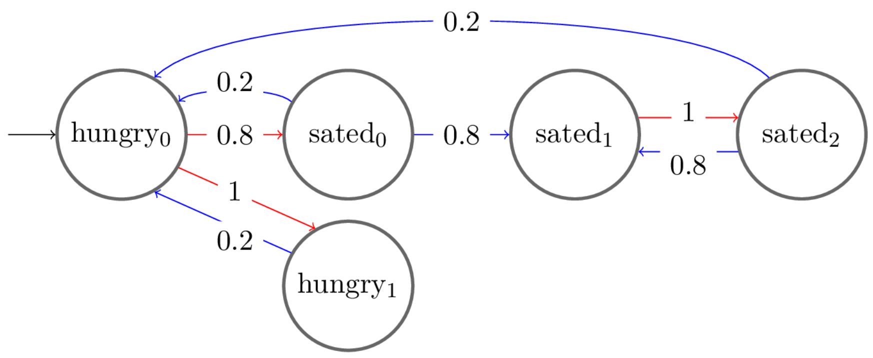


Figure 2.6: The MC derived from the MDP M in example 2.2.1

2.3 Partially Observable Markov Decision Processes

Partially observable Markov decision processes (POMDPs) offer a way of modeling systems with state uncertainty. In such systems the agent that interacts with it has only limited information about the state it currently is in. The limited information about the state is called an *observation*.

Compared to MDPs, in POMDPs the agent chooses to perform an action without having full knowledge of the surrounding environment. Tools that assist agents in choosing actions will be discussed further in the thesis.

Definition 5. Partially observable Markov decision processes (POMDPs) are defined as a tuple $\mathcal{M} = (M, Z, O)$ where $M = (S, s_0, Act, \mathcal{P})$ is the underlying MDP, Z is a finite set of observations, and $O : S \to Z$ is the observation function that assigns observations to actions.

In POMDPs, one observation, which can but does not have to be unique in the model, is assigned to each state. An observation that is assigned to more than one state is called *potentially imperfect* [11].

Example 2.3.1. Let's take the MDP from the previous example 2.2.1 and transform it to a POMDP by assigning each state an observation – if the baby is hungry, we observe that it's crying, if it's in the sated state, we see it smiling.

Solution. This POMDP is very straightforward – each state is uniquely defined by its observation and the caretaker – agent – can take appropriate actions to get the baby to its desired state.

Example 2.3.2. But what happens if two more states are added between *hungry* and *sated*? The new states are *almost hungry* and *almost sated* and if the baby is in these states it is neither crying nor smiling, the observation is *neutral*.

Solution. Figure 2.7 depicts the underlying MDP. In the *hungry* and *sated* states, it is perfectly clear which action should the caretaker take. A hungry baby must be fed while a sated baby can be ignored. The caretaker, however, cannot distinguish between the *almost hungry* and *almost sated* states and therefore cannot determine what action should be taken based on the observation alone. This is *state uncertainty* [11].

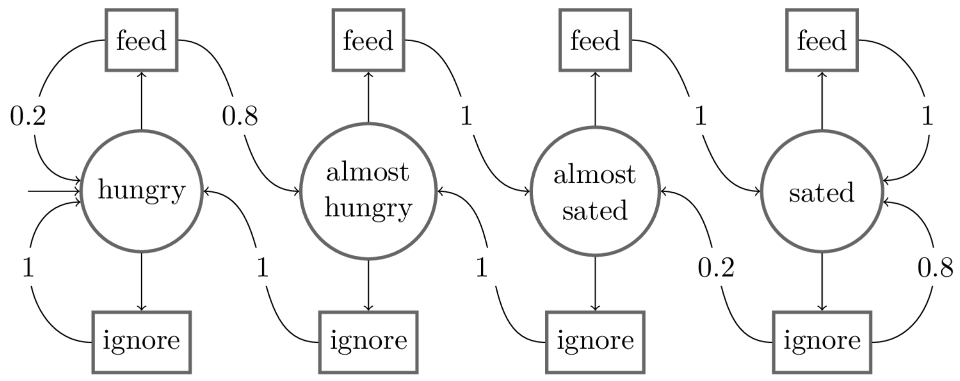


Figure 2.7: The underlying MDP representing the four-state baby.

Chapter 3

Existing Methods for Controlling POMDPs

This chapter discusses the two main approaches to state uncertainty in partially observable Markov decision processes – the *belief-based* approach and *controller synthesis*.

Definitions in this chapter are taken over and adapted from Algorithms for Decision Making [11] (definition 6), History-based controller design and optimization for partially observable MDPs [12] (definition 7), and from Shepherding Hordes of Markov Chains [15] (definitions 8, 9, 10, and 11)

3.1 Belief-Based

In each step the agent takes, a *belief* is calculated based on past actions and observations. This belief is represented by a probability distribution over the underlying states of the POMDP – the probability that the agent is currently in that state of the model [11].

The *initial belief* is the assumption made at the beginning. It can either be very generic (e.g. evenly distributed between all possible states) or, if there is additional information available at the start, it can be more specific to a subset of the states. However, making the initial belief too specific to a certain area of the POMDP brings a considerable risk – if the initial belief is wrong, the decisions, and as a result the whole path will be skewed [11].

After every action taken by the agent – based on the observation it receives and the belief distribution – the belief is updated using one of the various methods such as the discrete state filter.

Discrete State filter

If the number of states and observations in a POMDP is finite, we can use the *discrete state filter*, a recursive bayesian estimation method, for calculating the belief distribution using the current observation and the action that was taken [11].

Definition 6. In a POMDP where S is the set of states, Act is the set of actions, and Z is the set of observations, then B will be the belief space where there is a belief $b(s) \geq 0$ for all $s \in S$. The sum of all $b(s) \in B$ is 1.

$O(o|a, s')$ is the probability of observing observation $o \in Z$ after taking action $a \in Act$ and transitioning to state $s' \in S$. $T(s'|s, a)$ is the probability that the model transitioned to state $s' \in S$ after taking action $a \in Act$ in state $s \in S$.

The new belief $b'(s')$ can be the calculated as follows:

$$\begin{aligned}
b'(s') &= P(s'|b, a, o) && (3.1)\\
&\propto P(o|b, a, s')P(s'|b, a) && (3.2)\\
&= O(o|a, s')P(s'|b, a) && (3.3)\\
&= O(o|a, s')\sum_{s} P(s'|a, b, s)P(s|b, a) && (3.4)\\
&= O(o|a, s')\sum_{s} T(s'|s, a)b(s) && (3.5)
\end{aligned}$$

The more accurate the observations and transition model are, the more successful the belief update is [11].

Example 3.1.1. Let's go back to the four-state baby POMDP from example 2.3.2 with states $S = \{h, ah, as, s\}$[1], actions $Act = \{feed, ignore\}$, and observations $Z = \{crying, neutral, smiling\}$. What would be the belief space after the first two steps given that the initial belief space is an equal distribution?

Solution. Using formula 3.5 we can calculate all potential belief spaces for each state, action, and observation.

States h and s are not imperfect and can be transitioned to as a result of both actions. Therefore, regardless of the action the caretaker takes, the probability of observing the observations these states are associated with (h – *crying*, s – *smiling*) is equal to 1 and the probability of observing the other two observations is 0. States ah and as are associated with the same observation (*neutral*) and can be transitioned to with both actions, therefore $O(neutral|a, ah) = O(neutral|a, as) = 0.5$ for $\forall a \in Act$ and 0 for the other two observations.

We start with an equal distribution, therefore $b(h) = b(ah) = b(as) = b(s) = 0.25$.

Table 3.1 depicts the potential belief updates for all states, actions, and observations after the first step.

Action	feed						ignore					
State / Observation	neutral		crying		smiling		neutral		crying		smiling	
$b'(h)$	0	0	0.05	1	0	0	0	0	0.50	1	0	0
$b'(ah)$	0.10	0.44	0	0	0	0	0.13	0.83	0	0	0	0
$b'(as)$	0.13	0.56	0	0	0	0	0.03	0.17	0	0	0	0
$b'(s)$	0	0	0	0	0.05	1	0	0	0	0	0.25	1

Table 3.1: Potential belief updates before (left) and after normalization (right).

Table 3.1 depicts the potential belief updates for all states, actions, and observations after the second step given that the agent chose to *feed* the baby and received a *neutral* observation after that.

[1] h = hungry, ah = almost hungry, as = almost sated, s = sated

Action	feed						ignore					
State / Observation	neutral		crying		smiling		neutral		crying		smiling	
$b'(h)$	0	0	0	0	0	0	0	0	0.44	1	0	0
$b'(ah)$	0	0	0	0	0	0	0.28	1	0	0	0	0
$b'(as)$	0.22	1	0	0	0	0	0	0	0	0	0	0
$b'(s)$	0	0	0	0	0	0	0	0	0	0	0.11	1

Table 3.2: : Potential belief updates before (left) and after normalization – $b(ah) = 0.44$, $b(as) = 0.56$

Table 3.3 depicts the potential belief updates for all states, actions, and observations after the seconds step if the first action taken by the agent was *ignore*.

action	feed						ignore					
state / observation	neutral		crying		smiling		neutral		crying		smiling	
$b'(h)$	0	0	0	0	0	0	0	0	0.83	1	0	0
$b'(ah)$	0	0	0	0	0	0	0.08	1	0	0	0	0
$b'(as)$	0.42	1	0	0	0	0	0	0	0	0	0	0
$b'(s)$	0	0	0	0	0	0	0	0	0	0	0.03	1

Table 3.3: : Potential belief updates before (left) and after normalization – $b(ah) = 0.83$, $b(as) = 0.17$

In this example, we see that it takes two steps at most for the belief state to be equal to 1 for a particular state.

3.2 Finite-State Controller Synthesis

Unlike the belief-based approaches, which need to calculate the new belief state after every step that the agent takes, using a controller reduces the complexity of determining the next action during the run to what essentially is just a table lookup. The process of programmatically finding a finite-state controller for a particular POMDP is called *controller synthesis*.

3.2.1 Finite-state Controllers

Definition 7. A *finite-state controller* (FSC) for a POMDP $\mathcal{M} = (M, Z, O)$ (as established in definition 5) is a tuple $F = (N, \phi, \psi)$ where N is a finite set of *nodes*, function $\phi : N \to Act$ assigns an action to each node, and function $\psi : N \times Z \to N$ maps the current node and an observation to the next node.

This thesis will only consider deterministic finite-state controllers where the current node and the received observation determine the one next node. Stochastic FSCs which define the next node as a distribution over a subset of nodes will not be considered or discussed.

The process of controlling a agent's interactions with a POMDP using a FSC (also depicted by figure 3.1) has three basic phases that continuously repeat:

1. The agent receives an observation $z \in Z$ from a POMDP $\mathcal{M}$ and relays this information to the controller F.
2. The controller transitions to a new node n_{i+1} based on the current node n_i and the observation z it received. The new node n_{i+1} determines the action a.
3. Agent performs the action which causes the POMDP to transition to the next state with the next observation.

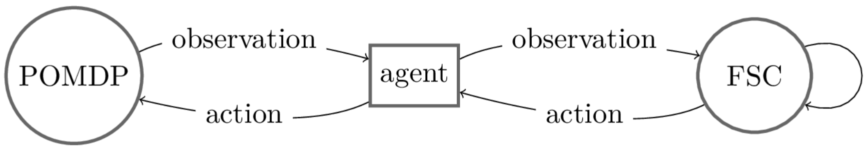
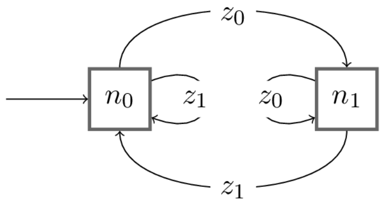


Figure 3.1: The process of controlling a POMDP with a FSC.

Example 3.2.1. POMDP $\mathcal{M}$ is defined by a set of states $S = \{s_0, s_1, s_2, s_3\}$ where in each state there are two possible actions (a_0 and a_1) which both cause a transition to one different state. Finite-state controller F has two nodes $N = \{n_0, n_1\}$, node-to-action mapping $\phi(n_0) = a_0, \phi(n_1) = a_1$, and $\psi(n_{0,1}) = \begin{cases} n_1 & \text{if } z_0 \\ n_0 & \text{if } z_1 \end{cases}$ defines the controller transitions based on the received observation. s_0 and n_0 are the starting state and node. FSC F controls the POMDP $\mathcal{M}$ and both are depicted in figure 3.2. How does the path an agent takes in this POMDP change based on how observations z_0 and z_1 are assigned to states s_0, s_1, s_2, and s_3?



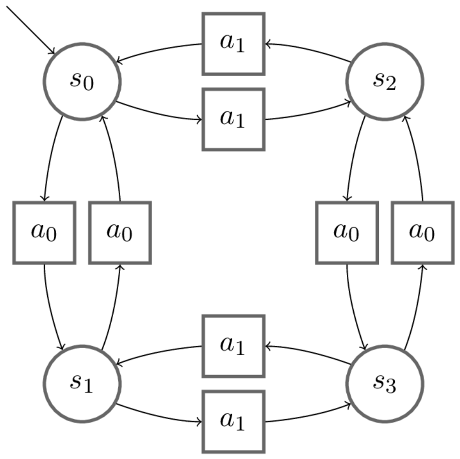


Figure 3.2: Finite-state controller F (left), POMDP $\mathcal{M}$ (right)

Solution. Depending on the observation assignment to the states, the agent navigating POMDP $\mathcal{M}$ controlled by the FSC F will behave in one of the following ways:

1. If $O(s_0) = O(s_3) = 0$ and $O(s_1) = O(s_2) = 1$ results in the agent traveling in an anti-clockwise circle $s_0 s_1 s_2 s_3 s_0 ...$; if $O(s_0) = O(s_3) = 1$ and $O(s_1) = O(s_2) = 0$ the agent will travel in a clock wise circle $s_0 s_3 s_2 s_1 s_0 ...$. In both cases, the controller alternates between both nodes.

2. If $O(s_0) = O(s_3) = 0$ or $O(s_0) = O(s_1) = 1$ – the agent will oscillate between the two states (s_0 and s_3 or s_0 and s_1) with the FSC ending up in node n_0 for observation z_1 or n_1 or observation z_0, observations in the remaining two states are not important as the agent will never visit them.

3. If $O(s_0) = 1$ and $O(s_1) = O(s_2) = 0$ or if $O(s_0) = 0$ and $O(s_2) = O(s_3) = 1$ (the observation in the fourth state is not important) – after the first step away from s_0, the agent alternates between the other two states each step.

4. If $O(s_0) = O(s_1) = O(s_2) = 0$ and $O(s_3) = 1$ or if $O(s_0) = O(s_2) = O(s_3) = 1$ and $O(s_2) = 0$ – in this case the agent takes two steps before it starts alternating between states s_3 and s_1 or states s_3 and s_2.

History-Based Controllers

History-based controllers add more meaning to their controller nodes other than just the corresponding action [12]. FSCs with memory nodes a type of such controllers [1].

3.2.2 Controller Synthesis

If a finite-state controller can be modeled as a Markov chain, then a number of FSCs with the same set of nodes can be modeled as a family of MCs. In this section I present two of the existing synthesis methods for finite-state controllers.

Abstraction Refinement

Abstraction refinement (AR) is a method for evaluating large families of MCs and finding realisations that satisfy a certain specification introduced by Češka et. al in Shepherding Hordes of Markov Chains [15]. The specification (φ) in question is a quantitative property (e.g. reachability probability or expected reward acquired by visiting certain states) usually accompanied by a threshold (λ).

To determine which realisations of a MC family satisfy the specification and to avoid having to examine each realisation separately (which is ineffective), the AR approach first transforms the MC family into what is called an *all-in-one MDP* and then abstracts that into a more compact model called *quotient MDP*. A model checker such as Storm (further described in A Storm is Coming: A Modern Probabilistic Model Checker [6]) is then used to evaluate the quotient MDP by obtaining the values of p_{min} and p_{max}, under- and over-approximations of the model checking results, and comparing them to the threshold. Based on this information it can be determined whether all of the realisations encompassed in this quotient MDP satisfy φ, violate φ, or whether the family needs to be split into subfamilies which are then reexamined separately.

Definition 8. The *all-in-one MDP* of a Markov chain family $\mathfrak{D} = (S, s_0, K, \mathfrak{P})$ is defined as $M^{\mathfrak{D}} = (S^{\mathfrak{D}}, s_0^{\mathfrak{D}}, Act^{\mathfrak{D}}, \mathcal{P}^{\mathfrak{D}})$ where $S^{\mathfrak{D}} = S \times \mathcal{R} \bigcup \{s_0^{\mathfrak{D}}\}$ is the set of states, $Act^{\mathfrak{D}} = \{a^r | r \in \mathcal{R}^{\mathfrak{D}}\}$ is the set of actions, and $\mathcal{P}^{\mathfrak{D}}(s_0^{\mathfrak{D}}, a^r)((s_0, r)) = 1$ and $\mathcal{P}^{\mathfrak{D}}((s, r), a^r)((s', r)) = \mathfrak{P}(r)(s)(s')$ is the transition function.

Depicted in figure 3.3 is an example of a Markov chain family with one parameter a with the domain $T_a = \{s_1, s_2\}$ and transition probabilities $P(s_0) = a$, $P(s_1) = 0.5s_0 + 0.5a$, $P(s_2) = s_1$. (Transition probabilities are not included for the sake of readability.) This family of MCs will be used as the basis for the subsequent demonstrations of the all-in-one and quotient MDPs.

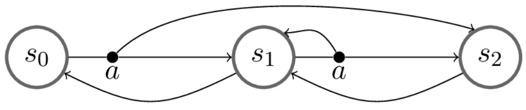


Figure 3.3: A MC family with parameter a.

Figure 3.4 depicts the all-in-one MDP constructed from realisations $r_1(a) = s_1$ and $r_2(a) = s_2$. The action taken in the initial state $s_0^{\mathfrak{D}}$ decides in which realisation the model operates thereafter and the states are labeled in such a way that it is possible to determine the corresponding realisation. Model checking this MDP yields information about all its states and therefore about all the realisations it encompasses, but for bigger MC families the all-in-one MDP can be too large to examine.

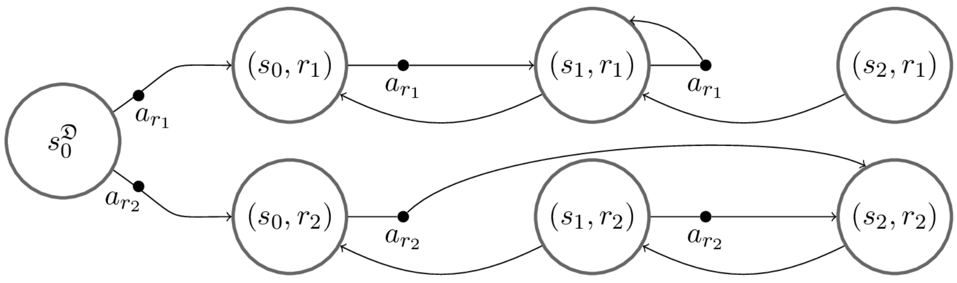


Figure 3.4: An all-in-one MDP that encompasses realisations r_1 and r_2.

A *quotient MDP* is a more compact model that is created by *abstracting* the realisation part from the state labels in the all-in-one MDP and therefore *forgetting* in which realisation it operates. Model checking this MDP also yields information about all is states and the realisations it encompasses.

Definition 9. *Forgetting* is an equivalence relation $\sim_f \subseteq S^{\mathfrak{D}} \times S^{\mathfrak{D}}$ that satisfies $(s, r) \sim_f (s', r') \iff s = s'$ and $s_0^{\mathfrak{D}} \sim_f (s_0^{\mathfrak{D}}, r) \forall r \in \mathcal{R}$. Forgetting induces the *quotient MDP* $M_{\sim}^{\mathfrak{D}} = (S_{\sim}^{\mathfrak{D}}, [s_0^{\mathfrak{D}}]_{\sim}, Act^{\mathfrak{D}}, \mathcal{P}_{\sim}^{\mathfrak{D}})$, where $\mathcal{P}_{\sim}^{\mathfrak{D}}([s]_{\sim}, a_r)([s']_{\sim}) = \mathfrak{P}(r)(s)(s')$.

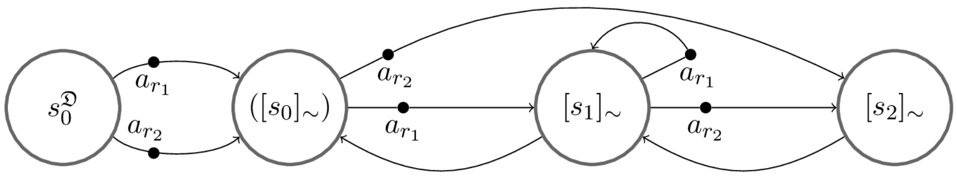


Figure 3.5: An abstraction over the all-in-one MDP depicted in 3.4 creates a quotiend MDP.

Definition 10. Splitting – $\mathfrak{D}$ is a family of MCs, $\mathcal{R} \subseteq \mathcal{R}^{\mathfrak{D}}$ is a set of realisations. For $k \in K$ and predicate A_k over S, splitting partitions $\mathcal{R}$ into
$\mathcal{R}_\top = \{r \in \mathcal{R} | A_k(r(k))\}$ and $\mathcal{R}_\bot = \{r \in \mathcal{R} |_k(r(k))\}$.

Definition 11. Restricting – $M^{\mathfrak{D}}_{\sim} = (S^{\mathfrak{D}}_{\sim}, [s^{\mathfrak{d}}_0]_{\sim}, Act^{\mathfrak{F}}, \mathcal{P}^{\mathfrak{D}}_{\sim})$ is the quotient MDP and $\mathcal{R} \subseteq \mathcal{R}^{\mathfrak{D}}$ a set of realisations. The restriction of $M^{\mathfrak{D}}_{\sim}$ with regards to $\mathcal{R}^{\mathfrak{D}}$ is the MDP $M^{\mathfrak{R}}_{\sim}[\mathcal{R}] = (S^{\mathfrak{D}}_{\sim}, [s^{\mathfrak{d}}_0]_{\sim}, Act^{\mathfrak{D}}[\mathcal{R}], \mathcal{P}^{\mathfrak{D}}_{\sim})$ where $Act^{\mathfrak{D}}[\mathcal{R}] = \{a_r | r \in \mathcal{R}\}$.

By splitting the set of realisations into two subsets and then restricting the quotient MDP using the subsets, we acquire two new quotient MDPs. Figure 3.6 depicts a smaller quotient MDP created by restricting the quotient MDP from figure 3.5 with the subset $\{r_1\} \subset \mathcal{R}^{\mathfrak{D}}$.

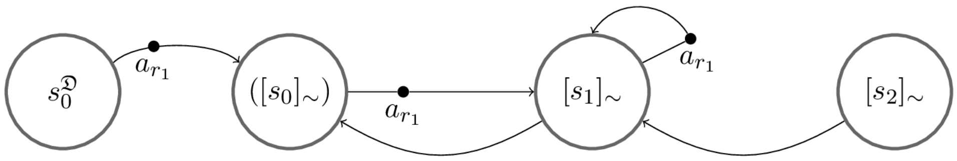


Figure 3.6: Restricted quotient MDP.

Threshold synthesis, as described in Shepherding Hordes of Markov Chains [15], is the part of the AR method that is tasked with partitioning realisations of a MC family into two subsets – one in which all realisations satisfy φ and one in which all realisations violate it.

The process starts with U, a set of sets of realisations that have not been examined yet, and the initial quotient MDP $M^{\mathfrak{D}}_{\sim}$. In the first step of the synthesis loop a set of realisations $\mathcal{R}$ is selected from U and the quotient MDP is restricted with regards to $\mathcal{R}$. Next a model checker is used to obtain the under- and over-approximations p_{min} and p_{max} of the probabilities or rewards for the given specification φ. Based on these, the algorithm can decide in which category the set of realisations belongs.

If $p_{max} \leq \lambda$, then all realisations satisfy φ, if $p_{min} > \lambda$, then no realisation in the set satisfies φ, and if $p_{min} \leq \lambda < p_{max}$ then it cannot be decided and the $\mathcal{R}$ needs to be split and the new sets of realisations are added to U.

The algorithm repeats until U is empty.

Optimum synthesis is a similar algorithm that is used to find the one realisation that best satisfies the specification. Like threshold synthesis it starts with a set of sets of realizations U, a quotient MDP $M^{\mathfrak{D}}_{\sim}$, and a variable max that holds the current optimal value.

Again, the synthesis process takes the sets of realisations from U and uses them to restrict the quotient MDP $M^{\mathfrak{D}}_{\sim}$ which is then examined by the model checker to determine values p_{min}, p_{max} and the corresponding scheduler.

If $p_{max} < max$ then the realisation is disregarded as there have already been more optimal schedulers. Otherwise, it checks if the synthesized scheduler is consistent, in that case it updates the values of max and marks this scheduler as optimal for the time being. If the scheduler is not consistent the algorithm further checks if $p_{min} > max$ and if that is the case, the value of max is still updated to p_{min}. In any way, if the scheduler is determined not consistent, the realisation $\mathcal{R}$ is split according to an appropriate predicate and both sets are added to U [15].

Counterexample-Guided Inductive Synthesis

Counterexample-guided inductive synthesis (CEGIS), introduced in Inductive Synthesis for Probabilistic Programs Reaches New Horizons by Andriushchenko et. al. [2], is a different method for examining families of Markov chains and finding realisations that satisfy a given specification.

There are two figurative parts to this method – a *learner* and an *oracle*. The learner maintains $\mathcal{Q}$, a set of realisations of the family of MCs that are to be checked. At the start of the process, the learner selects a realisation $r \in \mathcal{Q}$ and passes it to the oracle. The oracle verifies the realisation with respect to the specification φ and determines whether $r \models \varphi$.

Counterexamples are derived from realisations that have been verified, and rejected. Based on these, other realisations in $\mathcal{Q}$ that would also violate the specification are rejected by the learner and do not need to be verified by the model checker.

Figure 3.7 depicts the synthesis process and the figurative communications between the learner and the oracle.

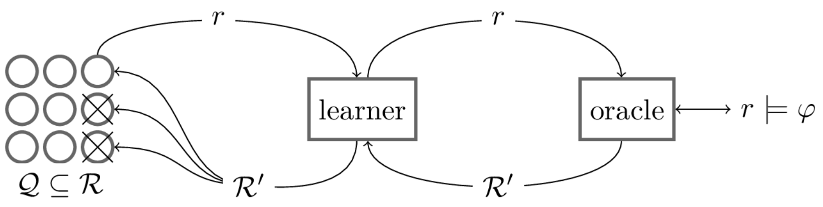


Figure 3.7: Graphical representation of the CEGIS synthesis method

Chapter 4

PAYNT – Probabilistic progrAm sYNThesizer

PAYNT, the Probabilistic progrAm sYNThesizer, is a Python program by Andriushchenko et al. introduced in PAYNT: A Tool for Inductive Synthesis of Probabilistic Programs [3] and its purpose originally, as the name suggests, was to synthesize solutions to probabilistic programs.

4.1 PAYNT for probabilistic programs

Probabilistic programs provide a formal way to describe systems that deal with some kind of uncertainty [5]. A *sketch* models a family of probability programs using the PRISM language. Among these programs, PAYNT searches for the one that best satisfies the given *specification* (usually reachability property or expected reward) and *synthesis problem* (minimization or maximization).

Sketches define all states, transitions and actions of the program. They also contain undefined parameters called *holes*. "Filling" the holes with options from specified domains creates realisations that can be verified against the specification using the model checking tool Storm [6].

The challenge here is to evaluate all possible realisations in a timely manner. Because number of realisations grows exponentially with the number of holes, the simple synthesis method *one-by-one* quickly becomes too slow. For these purposes, PAYNT implements the synthesis methods from the previous chapter 3 – AR and CEGIS, and their combination called Hybrid.

4.2 PAYNT for POMDPs

For Inductive Synthesis of Finite-State Controllers for POMDPs [1], PAYNT was adapted to be able to synthesize deterministic finite-state controllers for POMDPs.

In this case, there are no holes in the input sketch, instead it represents the POMDP for which PAYNT is trying to find the optimal controller. The sketch defines all states, actions, transition functions, and observations of the POMDP.

There are two figurative parts to the controller synthesis process: the *outer stage* – where the *design space* of a set memory size is created – and the *inner stage* – where the design space is explored [1].

The design space represents all possible FSCs that PAYNT has to consider. *Memory size* defines how many different memory values should the final FSC have. In the initial design space, all actions and observations are available in all memory value nodes.

For design space exploration in the inner stage, PAYNT again uses one of the implemented synthesis methods (e.g. AR) to find the optimal controller in the design space.

4.2.1 Design Space

In the program, the design space is represented by a list of *Holes*, Python classes, with properties such as `name`, `options`, and `option_labels`.

Hole names are strings that encode the memory values of the FSCs, as well as the observations that are received by the agent from the POMDP. They have a set structure of `"T([O], M)"`, where `T` defines the hole type – either `A` for *action*, `M` for *memory*, or `AM` for the combination of both action and memory – `O` stands for the observation which is represented by a string, and `M` is the numerical memory value of the node.

The Hole `options` instance attribute is a list of integers that represent a certain possibility of what could happen next if the received observation is as defined in the hole name. Depending on the hole type, that could be the *action* that should be taken, a *memory update*, or a combination of both.

The hole `option_labels` attribute provides further information about the options that are human readable. Each hole option has a option_label that describes it.

4.2.2 Iterative Strategy

PAYNT can continuously increase the memory size and search for finite-state controllers in an increasingly larger design space.By keeping the current optimum values this strategy ensures that only FSCs that improve this optimum are accepted in the synthesis process. This strategy will be referred to as the *iterative* approach.

With this strategy, PAYNT is guaranteed to find the best available FSC for each memory size, however as it increases, the design space also grows in size and the synthesis takes longer.

4.2.3 Memory Injection Strategy

Andriushchenko et al. in [1] introduce a strategy for finding compact FSCs by using information from the previous inner state loop for adding memory values to selected observations and then *removing symmetries* to reduce the design space size and remove FSCs with the same value.

In this thesis, this strategy will be referred to as the *memory injection* or just *injection* approach.

Chapter 5

Contributions to PAYNT

In this chapter, I describe my contributions to PAYNT – their objective, design and implementation.

5.1 Graphical FSC Representation

So far, the finite-state controller output by PAYNT was always in a text format. While this format is human readable, it is difficult to get an understanding of how the controller is shaped or how the transitions between nodes work. Therefore, my first contribution to PAYNT was to figure out a way to programmatically output a graphical representation of the synthesised controllers (or generally any design spaces).

5.1.1 Implementation

For visualizing the controllers, I chose to use PyGraphviz [8], an open-source Python interface for drawing graphs build on the Graphviz [7] visualization software.

In the code, the design space is stored in a list of `Holes`, nodes in the controllers. The class variable `Hole.name` encodes the current memory value and incoming observation, `Hole.options` holds a list of integers that identify the node that could be transitioned to, and `Hole.option_labels` provides further information about the next node and possibly the action that the agent is supposed to take.

For determining the memory value from the `Hole.name`, I implemented a short function that uses a regular expression to match the memory value inside the hole name:

```
def get_current_memory(name):
    return int(re.findall(r"[AM]{1,2}\(\[.*\],(\d+)\)", name)[0])
```

A similar function matches the observation inside the hole name:

```
def get_current_observation(name):
    return re.findall(r"[AM]{1,2}\(\[(.*)],\d+\)", name)
```

Because in some cases the observation is an empty string and the list returned by the `findall()` function would be empty, I cannot return the first element like in the `get_current_memory()` function. Instead, I chose to return the list itself and let it be addressed in a way that is appropriate for the context in which this function was called.

Next are the hole options. In case of pure action or memory type holes, the hole options do not need any special treatment and can be used as they are. But in case of the hole type which combines action and memory together, the hole options do not match the memory value and instead the memory value needs to be parsed from the corresponding option label. This was implemented in the following function:

```
def parse_am_labels(hole):
    options = []
    for option in hole.options:
        opt = re.findall(r"{.*}\+(\d+)", hole.option_labels[option])[0]
        options.append(int(opt))
    max = None if not options else sorted(list(set(options)))[-1]
    return (options, max)
```

In the end, I transform the parsed data into a nested dictionary in this format – the memory values, nodes, are the keys in this dictionary:

```
{
    start_node: {
        end_node1: [observation1, observation3],
        end_node2: [observation2]
    },
    end_node2: [...],
    ...
}
```

This allows to then iterate over this dictionary and add nodes and edges to the Py-Graphviz `AGraph` object using methods `AGraph.add_nodes_from` and `add_edge`.

The final graph is then output in the form of a PNG image to a specified file. For examples of these images, see appendix A.

5.2 Limiting Design Space

With no additional logic, the design space from which PAYNT synthesizes the optimal finite-state controller contains all possible combinations of memory values, observations, actions and transitions. When the memory size is increased, the design space size also increases and so does the synthesis duration. If only certain transitions were allowed, it could reduce the synthesis time and allow finding controllers with memory sizes that otherwise might have been unobtainable in a reasonable time.

5.2.1 Implementation

For restricting the design space, I implemented the `set_memory` function which takes the design space, memory size, a *condition* function, and further function arguments to control the behaviour of the function – `rewrite` and `restrict`. The condition function can be any function that takes three numerical arguments that represent the `current` memory value, the `next` memory value, and the maximum (`max`) memory value and returns a boolean.

This function iterates over all of the holes in the design space and extracts the current memory value from the hole name using the `get_current_memory` function described in section 5.1. Then, new options are constructed based on the current memory, memory size and the condition function:

```
new_options = [next for next in hole.options
    if not condition(current, next, mem_size-1)] if restrict
    else [next for next in range(mem_size) if
    condition(current, next, mem_size-1)
```

The `restrict` parameter controls whether the new options come from a generated range of numbers that are selected if the condition evaluates true (`restrict == False`) or if the new options are based on the existing options while removing those that pass the condition function (`restrict == True`).

If the hole is of type memory, the new options are added to (if `rewrite == False`) or replace the hole options. For mixed type holes, the new options which represent memory values need to be mapped to observation labels and the hole options then are extended or replaced with matching indexes of these observation labels.

The restrictions do not take actions or observations into account.

5.2.2 Restrictions

Because the number of possible restriction conditions is next to infinite, I have had to decide which restriction conditions to explore in this thesis. Or rather which formats of design spaces should PAYNT examine to synthesize the optimal FSC. For that, I have set myself the following conditions:

1. All nodes of the design space must be reachable from the initial node 0. Restricting the design space to be equal to a design space achievable with a smaller memory size is not desirable.
2. A valid FSC with at least one reachable absorbing state must be synthesizable from the restricted design space.
3. Transitions are only allowed between memory states equal, differing by one, or from smallest to largest and vice versa. This condition is due to personal choice of the FSCs I wanted to focus on.

Using different condition functions and parameters, the `set_memory` function is able to shape the design space to one of the following forms.

Forward

The *Forward* restriction produces the smallest possible design space that can still result in a valid controller. It is constructed by first clearing, emptying the design space using a condition function that always evaluates to `False` and allowing rewrite and then adding holes that pass the condition function `__forward`. The conditions allows transitions from nodes with memory values x_i to nodes with memory values x_{i+1} or from x_{max} to x_{max}. A graphical representation of the nodes and available transitions is shown in figure 5.1.

```
def __forward(self, current, next, max):
    return current + 1 == next or (current == max and next == max)
```

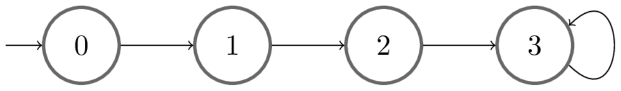


Figure 5.1: Restriction Forward

One step

The restriction *One step* builds on the design space previously constructed using the Forward restriction. Using the `set_memory` function with the following condition `__backward` it adds transitions from nodes with memory values x_i to nodes with memory values x_{i-1} or from x_0 to x_0. The resulting design space can be seen in figure 5.2.

```
def __backward(self, current, next, _):
    return current - 1 == next or (current == 0 and next == 0)
```

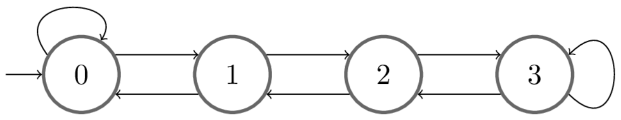


Figure 5.2: Restriction One step

Backward

To create the restriction *Backward*, the condition function `__self_loops` is used to remove transitions from the first and last nodes to themselves on a design space already restricted by the One step restriction.

```
def __self_loops(self, current, next, _):
    return current == next
```

A graphical representation of the resulting design space can be seen in figure 5.3.

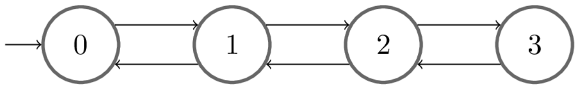


Figure 5.3: Restriction Backward

Simple circle

The restricted design space *Simple circle* is constructed using the `__simple_circle` condition function which only allows transitions from nodes with memory values x_i to x_{i+1} or from x_{max} to x_0. Figure 5.4 depicts a schema of the resulting design space.

```
def __simple_circle(self, current, next, max):
    return (current + 1 == next) or (current == max and next == 0)
```

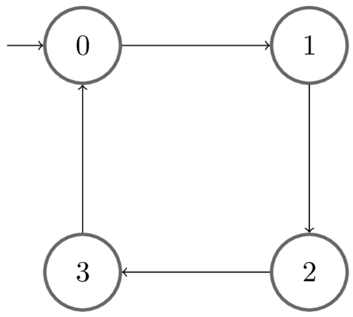


Figure 5.4: Restriction Simple circle

Circle both ways

The *Circle both ways* restriction builds on a Simple circle design space and adds transitions from nodes with memory values x_i to x_{i-1} and from x_0 to x_{max} using the function `__simple_circle_backward`. See figure 5.5 for a graphical schema of the resulting design space.

```
def __simple_circle_backward(self, current, next, max):
    return (current - 1 == next) or (current == 0 and next == max)
```

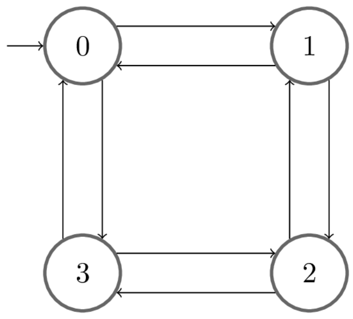


Figure 5.5: Restriction Circle both ways

Circle both ways with loops

Restriction *Circle both ways with loops* results in the biggest design space and it is constructed similarly to how Backwards is constructed from One step. Only this time, instead of using the `__self_loops` function to remove transitions from nodes with memory values

x_0 to x_0 and from x_{max} to x_{max}, it is used to add them. The design space that results from this restriction is depicted in figure 5.6.

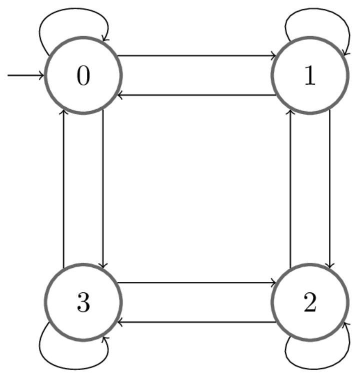


Figure 5.6: Restriction Circle both ways with loops

5.3 Incremental Memory Setting

The next step was to use the newly implemented way of restricting the design space to try to continuously synthesize FSCs from an increasingly larger design space. This includes using different restrictions as well as increasing the memory size.

5.3.1 Implementation

A new `SynthesizerPOMDPIncremental` class inherits from `SynthesizerPOMDP`. It uses SynthesizerPOMDPs initialization method and then sets three additional instance attributes – `memory_size` to set the initial memory size, `max_size` to set the maximum memory size that PAYNT should attempt to synthesize FSCs with, and `reset_optimum` to allow or prohibit resetting `Sketch.specification.optimality.optimum` to `None`. If this variable is not reset to `None`, it allows the synthesis loop to reject controllers that fall bellow this value faster.

In the `run` method of the SynthesizerPOMDPIncremental class, the `memory_size` and `max_size` attributes will be used in the synthesis loop. PAYNT will synthesize finite-state controllers until the gradually incremented `memory_size` attribute is equal to `max_size`. If the parameter `max_size` is set lower than the `min_size` parameter, the synthesis will run in an infinite loop.

For each memory size, the new memory size has to be set using the PomdpManager of the quotient attribute of the sketch and the memory then has to be unfolded – the design space is reset.

For each memory size, PAYNT then runs a sequence of restrictions, after each of those, it evaluates the design space size and (if it's greater than zero) PAYNT attempts to synthesise a finite-state controller. With the AR method of synthesis, the synthesizer only returns a FSC if it's able to find one with a better optimality value than previously.

Chapter 6

Evaluation

In this chapter I will evaluate the incremental memory setting approach against the baseline – iterative approach – and the memory injection approach; and I will compare these methods with regards to synthesis time, memory sizes, and calculated optimums.

6.1 Benchmarks and Experiment Conditions

The methods will be evaluated against two types of examples – Grid and Maze – and their variations Grid Avoid, Grid Center, and Maze Long.

All experiments run consecutively on an Acer TravelMate X laptop with an Intel® Core™ i5-8250U CPU @ 1.60GHz × 8 processor. For each benchmark, each approach was run for 30 minutes using the AR synthesis method.

6.1.1 Grid

The Grid is a simple example where a robot – agent – navigates in a 4x4 grid. The robot is able to move in directions *north*, *east*, *south*, and *west*. In case the robot is standing on the border of the grid and wants to take an action which would essentially lead out of the grid, the robot's action fails and it stays in the current field. The objective for the robot is to navigate to a set *target* field while avoiding potential *bad* fields.

The agent is placed in a random field in the grid with the *target* and *bad* fields not being among the options.

Each action the robot takes has a 90% chance of being successful. In the remaining 10%, the robot does not move and stays in its current field.

Grid Avoid

Grid Avoid, as depicted in figure 6.1a, has the target field T on coordinates $(x = 0, y = 3)$ and a bad field B on coordinates $(x = 1, y = 1)$ which the robot must try to avoid.

In each field, the robot receives robot receives one of four observations (0 – the initial field, 1 – any field inside the grid, 2 – the target field, and 3 – the bad field).

The robot receives a perfect observation on this field with the value of 3.

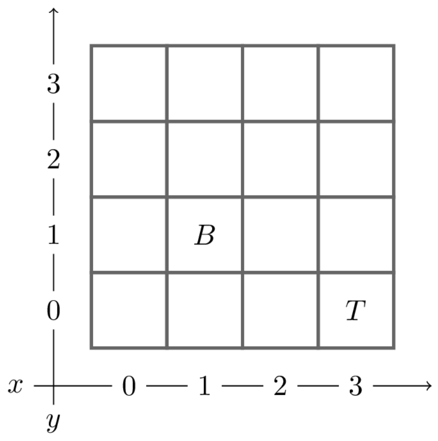
The quality of the controller is measured by the probability of the robot reaching the *target* field and not reaching the *bad* field.

Grid Center

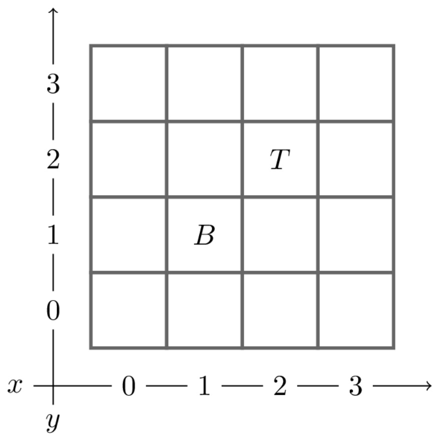
The Grid Center variation of the example (shown in figure 6.1b) moves the target field T to coordinates $(x = 2, y = 2)$.

What makes this challenge more difficult is that the robot now must take all of the four available actions *north*, *east*, *south*, and *west* to get to the target state. As opposed to the previous benchmarks where actions *south* and *east* were sufficient.

The quality of the controller is again measured by the probability of the robot reaching one of the end fields T and B.



(a) Grid Avoid



(b) Grid Center

Figure 6.1: Schemas of the Grid benchmarks with descriptive axes. T – target field, B – bad field

6.1.2 Maze

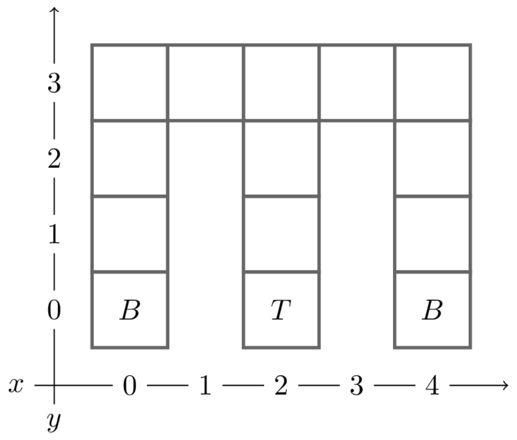
The Maze is a simple example used by Andriushchenko et al. in [1].

It again uses a robot that can go *up*, *down*, *left*, and *right* as its agent. This robot is not completely reliable, it is only successful in 80% of cases. For each action the robot tries to perform, there is a 16% chance that it ends up going in one of the perpendicular directions (8% chance for each of them) and a 4% chance that it will actually go in the exact opposite direction to where it intended to go.

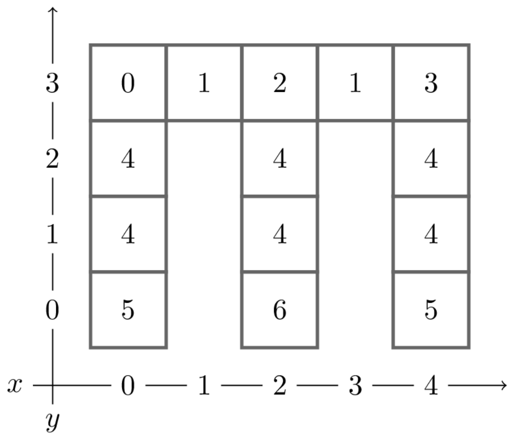
Regardless of the success of the robot's action, if the nature of the field does not allow the robot to move in that direction, the robot does not move at all and remains in its original field.

The observation in each field is consistent with the possible ways a robot could move from that field – e.g. a fields that allows the agent to move *up* and *down* but not *left* or *right* receive the same observation, it our case represented by the number 4. In this maze, there are seven different observations that the robot could receive. Figure 6.2 depicts the Maze and its fields with their corresponding observations.

The quality of the controller is measured by how many steps the robot needs to get to the target state T. Initially, robot is placed randomly in one of the fourteen fields.



(a) Maze schema showcasing the *target* state T and *bad* states B.



(b) Maze schema showcasing the different observations received in each field

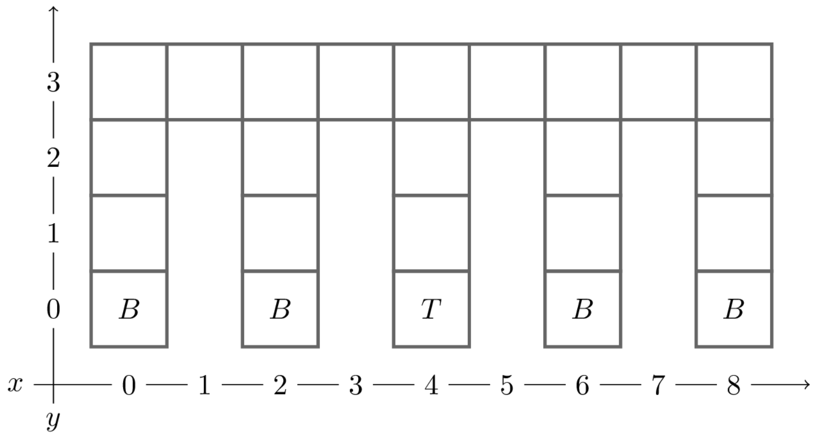
Figure 6.2: Two schemas of the Maze example with descriptive axes.

For observation 0, 2, 3, and 5, the action that needs to be taken for the robot to get to the Target field in the least amount of steps is very clear – observation 0 means that the robot will want to go right, observation 3 means the robot must try to go left. In observation 4 and 1, however, depending on the particular field, there are two correct actions.

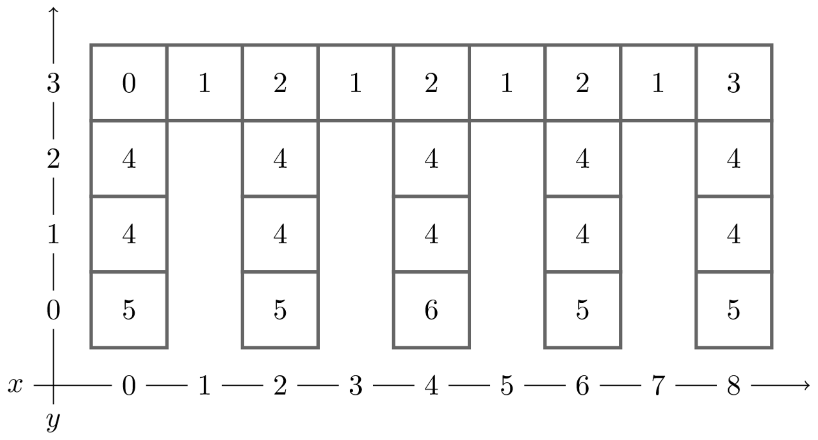
Maze Long

Maze Long is a variant of the example described in the previous section. It extends the maze to both sides and increases the number of fields with the critical observations 1 and 4 and also makes the observation 2 no longer automatically associated with the action *down*. A schema of this example is depicted in figure 6.3.

The agent is, too, initially placed in a random field and its objective is to get to the target field in as little steps as possible.



(a) Maze Long schema showcasing the *target* state T and *bad* states B.



(b) Maze Long schema showcasing the different observations in each field.

Figure 6.3: Two schemas of the Maze Long example with descriptive axes.

6.2 Evaluations

6.2.1 Limiting Design Space

In this section I explore the results of controller synthesis for two different benchmarks – Grid Avoid and Maze – with and without restrictions.

Grid Avoid

Let's take the Grid Avoid example as the benchmark and compare the results of synthesis between the full design spaces and design spaces restricted using different conditions.

As described in section 6.1.1, this benchmark measures the optimality of the controllers by the probability that the robot will be able to get to the target state T while not landing on the *bad* state B. The higher and closer to 1 is the *Optimum* value, the better is the controller.

Table 6.1 shows the optimum values of controllers acquired by exploring variously restricted design spaces with memory sizes 3-5 and how long PAYNT took to synthesize those controllers.

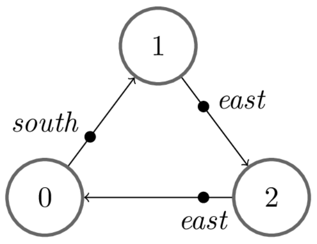
In this example, PAYNT is able to find a controller in each of the various design spaces. However, the controller quality can be vastly different depending on the applied restriction. Some restrictions (e.g. Simple circle) are able to find controllers of the same quality as the unrestricted design space and do that in a fraction of the time, while other restrictions (e.g. Forward) produce controllers of inferior quality.

The restriction predicates Circle both ways and Circle both ways with loops restrict to unnecessarily large design spaces, *Simple circle* is sufficient enough in this case.

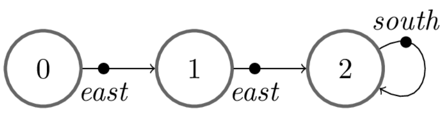
Restriction	Memory size	Optimum	Synthesis duration [s]
None	3.0	0.892274	0.353155
Backward	3.0	0.846731	0.027576
Circle both ways	3.0	0.892273	0.151318
Circle both ways with loops [1]	3.0	0.892274	0.369734
Forward	3.0	0.385714	0.003453
One step	3.0	0.880092	0.100280
Simple circle	3.0	0.892274	0.034246
None	4.0	0.911625	12.525107
Backward	4.0	0.880092	0.059986
Circle both ways	4.0	0.911625	0.917087
Circle both ways with loops	4.0	0.911625	3.379023
Forward	4.0	0.531429	0.004257
One step	4.0	0.902882	0.485129
Simple circle	4.0	0.911625	0.105396
None	5.0	0.920518	561.828769
Backward	5.0	0.902882	0.621761
Circle both ways	5.0	0.920518	4.734108
Circle both ways with loops	5.0	0.920518	29.257244
Forward	5.0	0.666857	0.048645
One step	5.0	0.915777	1.683551
Simple circle	5.0	0.920518	0.276828

Table 6.1: Results of FSC synthesis for the Grid Avoid example using different design space restrictions.

Figure 6.4 depicts the baseline FSC (6.4a) and the inferior FSC synthesized using the Forward restriction (6.4b) on design space with memory size 3 with their nodes and actions. The figures do not depict observations because actions upon receiving observations other than 1 do not need to be considered.



(a) FSC synthesized from a non-restricted design space



(b) FSC synthesized from a design space restricted using the Forwards predicate

Figure 6.4: FSC controllers with memory size 3 synthesized for the Grid Avoid benchmark

Figure 6.5 depicts the paths of two robots controlled by different FSCs starting in the field with coordinates $(0, 3)$.

[1]It is important to note that by its nature, this predicate does not restrict the design space for memory size 3.

The red robot, controlled by FSC depicted in 6.4a takes a path $\omega = (0,3)$, $(0,2)$, $(1,2)$, $(2,2)$, $(2,1)$, $(3,1)$, $(3,1)$, $(3,0)$, given that it does not "slip" at any point in time and all its actions are successful.

The path taken by the blue robot controlled by the FSC depicted in 6.4b is $\omega = (0,3)$, $(1,3)$, $(2,3)$, $(2,2)$, $(1,2)$, $(0,2)$, $(0,2)$,... With that initial state, this controller is not able to navigate its robot to the target field. In fact, a robot controlled by this FSC will not reach the target field T if the starting field is on with coordinate $x = 0$. This is why it has a low optimality value of 0.385714.

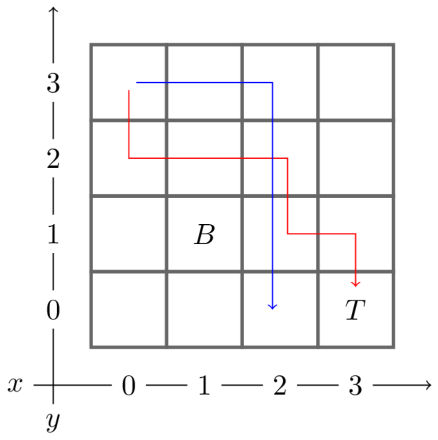
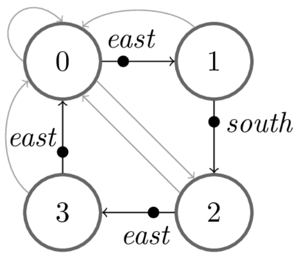
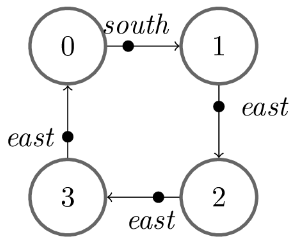


Figure 6.5: Path of the robots controlled by FSC depicted in 6.4a (red) and FSC depicted in 6.4b (blue)

Finite-state controllers depicted in figure 6.6 are the results of synthesis from a non-restricted design space with memory size 4 (6.6a) and a design space with the same memory size restricted using the *Simple circle* restriction (6.6b). These controllers have the same optimum value of 0.911625.



(a) FSC synthesized from a non-restricted design space



(b) FSC synthesized from a design space restricted using the Simple circle predicate

Figure 6.6: FSC controllers with memory size 4 synthesized for the Grid Avoid benchmark

Gray edges in 6.6a represent transitions upon receiving observations 0, 2, or 3. Observation 0 – outside of grid – is never received and observations 2, and 3 are not followed up by any action. Still, these transitions are a part of the controller, even if they do not influence the robot in any way on its path, and they have to be accounted for in the design space.

The time it takes PAYNT to synthesize a controller depends directly on the design space size. By applying a suitable restriction – in this case *Simple circle* is the best possible one, the optimal controller can be found 10 times faster (for memory size 3), 119 times faster (memory size 4), or even 2030 times faster (memory size 5).

Maze

The restrictions are less successful in the Maze example. Table 6.2 compares the synthesis results of synthesis applying different restrictions to the design spaces with memory sizes 3 and 4. In this example, the controllers with the lower optimum values are the ones with the better quality.

Restriction	Memory size	Optimum	Synthesis duration [s]
None	3.0	7.372105	16.960565
Backward	3.0	18.130855	1.271142
Circle both ways	3.0	16.277350	14.701138
Circle both ways with loops	3.0	7.372105	17.415230
Forward	3.0	71.266286	0.098195
One step	3.0	7.508047	4.559242
Simple circle	3.0	31.513882	0.663473
None	4.0	–	–
Backward	4.0	8.045141	0.702702
Circle both ways	4.0	7.552089	14.782590
Circle both ways with loops	4.0	7.372105	610.167898
Forward	4.0	67.010889	1.619117
One step	4.0	7.372105	62.611899
Simple circle	4.0	15.754049	2.317472

Table 6.2: Results of FSC synthesis for the Maze example using different design space restrictions

In this case, the FSCs synthesized by applying restrictions *Circle both ways with loops* (for memory size 3) and *One step* (for memory size 4) are the only controllers on-par with the best found FSC which was synthesized from the unrestricted design space with memory size 3.

However, just as with the previous benchmark, the *Circle both ways with loops* restriction does not restrict the design space for memory size 3 at all. Furthermore, synthesis from the memory size 4 design space that had the restriction *One step* applied takes 3.7 times longer than synthesis of the unrestricted design space with memory size 3 and this (restricted) larger memory design space does not produce controller of a better quality.

Because synthesis from the unrestricted design space with memory size 4 did not finish in the available time, it cannot be concluded whether if would produce a better quality controller or not. This leads to the following: either 7.372105 is the best optimality value we can expect a finite-state controller to have in this benchmark regardless of the memory size or the suitable restriction is not among the restrictions that were explored.

6.2.2 Incremental Memory Setting

The following sections show the results of controller synthesis for different examples using the three approaches – iterative, incremental, and memory injection.

Please note, that due to the nature of the benchmarks and the approaches, lines in graphs may overlap in places where the optimum value stagnates.

Maze

The Maze benchmark measures the optimality value of controllers by the ability of agents to get from a random initial field to the target field in as little steps as possible.

In section 6.2.1, it was established that PAYNT is not able to synthesize finite-state controllers with better optimality value than 7.372105 in the set time regardless of the memory size or if the design space was restricted in any way. By their nature, the *iterative* and *incremental* approaches cannot synthesize better controllers either.

Figure 6.7 shows how the controllers synthesized using the three different approaches compare in their optimality value with regards to used memory sizes. Overall, the results for this benchmark are quite unremarkable.

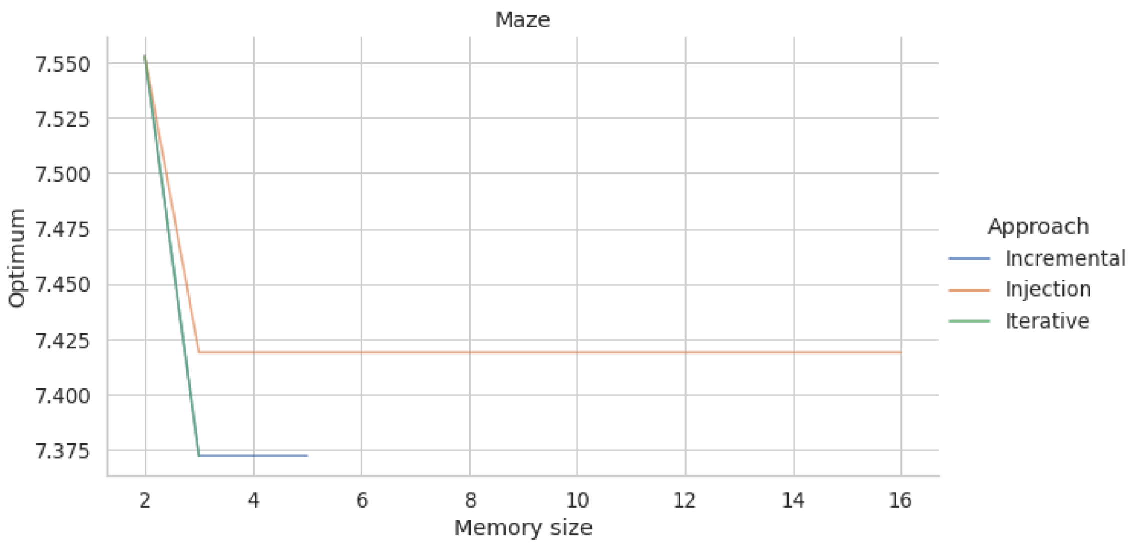


Figure 6.7: Graph comparing memory size and optimality value on the Maze benchmark.

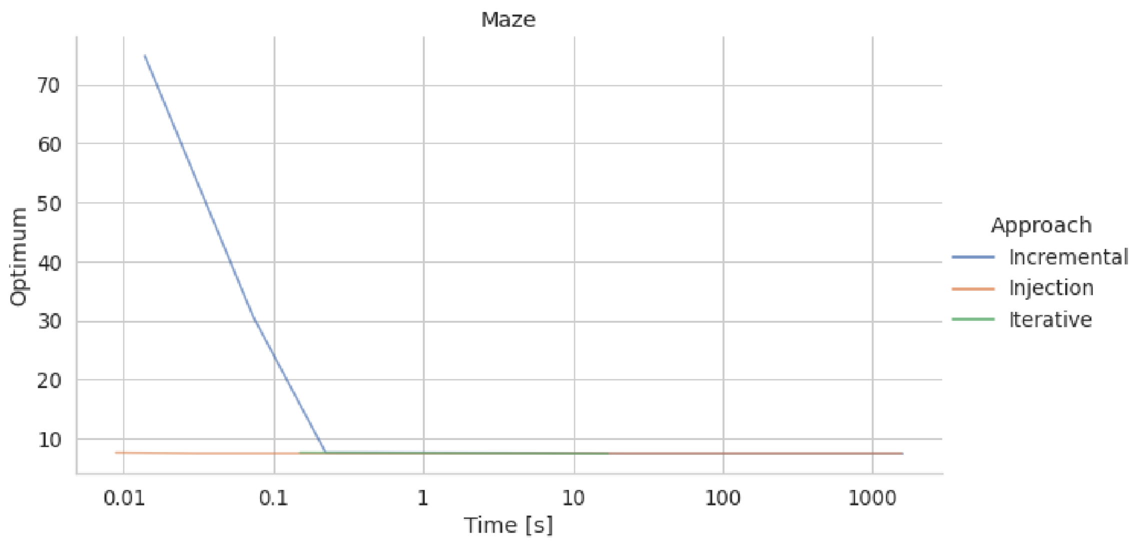


Figure 6.8: Graph comparing optimality value and time the Maze benchmark

Maze Long

Next, these approaches are evaluated on the Maze Long benchmark (described in section 6.1.2). The optimality values of the controllers found using the three different approaches with respect to available memory sizes are shown in figure 6.9. Figure 6.10 shows the results of all of the three approaches focusing on the optimality value of the found FSCs and total elapsed time.

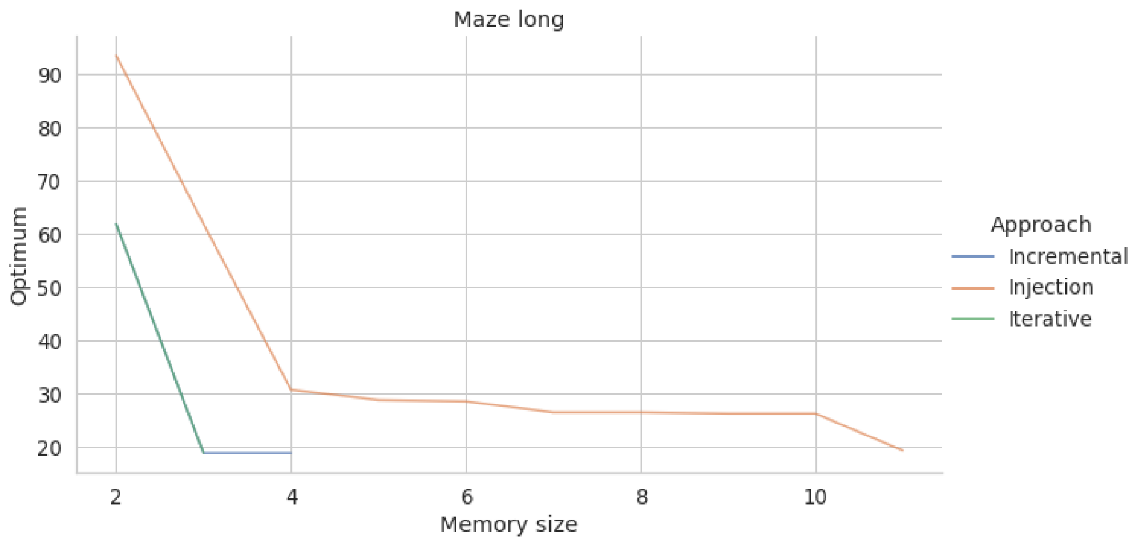


Figure 6.9: Graph comparing memory size and optimality value on the Maze Long benchmark.

The *iterative* approach is not able to synthesize controllers in design spaces with memory sizes 3 and above in the available time. The *incremental* approach does is able to find controllers of memory size 4 which is of better quality than the ones found by the *iterative* approach. The Memory injection approach finds controllers with memory sizes up to 10 but their optimality values are worse than of those found using the iterative and incremental approaches in the same time.

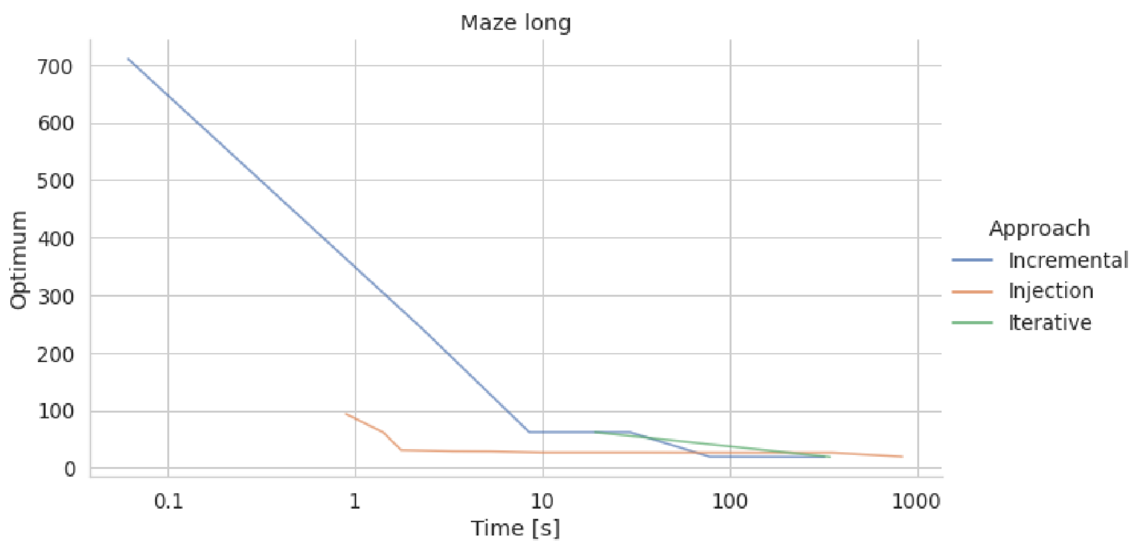


Figure 6.10: Graph comparing optimality value and time on the Maze Long benchmark.

Given that the longest straight path the robot could take in this maze is 9 steps long, the best found controller with optimality value 18.772289 is still far from that and is only a slight improvement from the 19.138456 found by the *iterative* approach.

Grid Avoid

Figure 6.11 shows the results of the three approaches and compares them with regards to optimality value and memory size. As established in section 6.2.1, restriction *Simple circle* seems to be able to restrict the design space in such a way that the controllers synthesized from it have the same quality as those synthesized from non-restricted design spaces. In this example, adding memory improves the quality of the controllers.

While the iterative strategy runs out of time by memory size 5, the incremental can continue to find controllers even past that point as the restrictions reduce the design space sizes and therefore the synthesis times as well. The memory injection strategy, however, does not seem to be able to find FSCs with improved optimality values after adding more than 4 memory values.

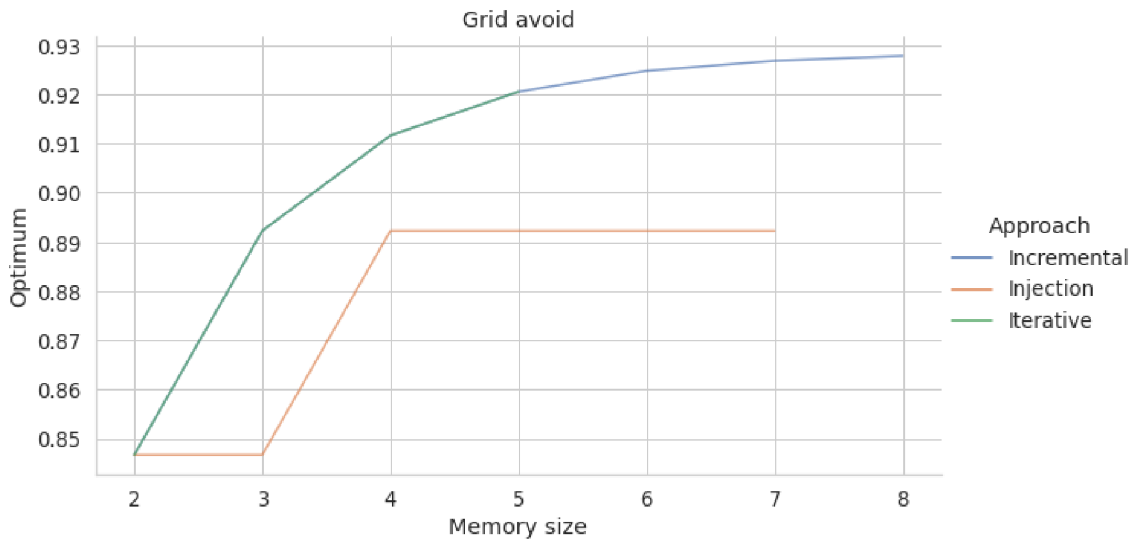


Figure 6.11: Graph comparing memory size and optimality value on the Grid Avoid benchmark.

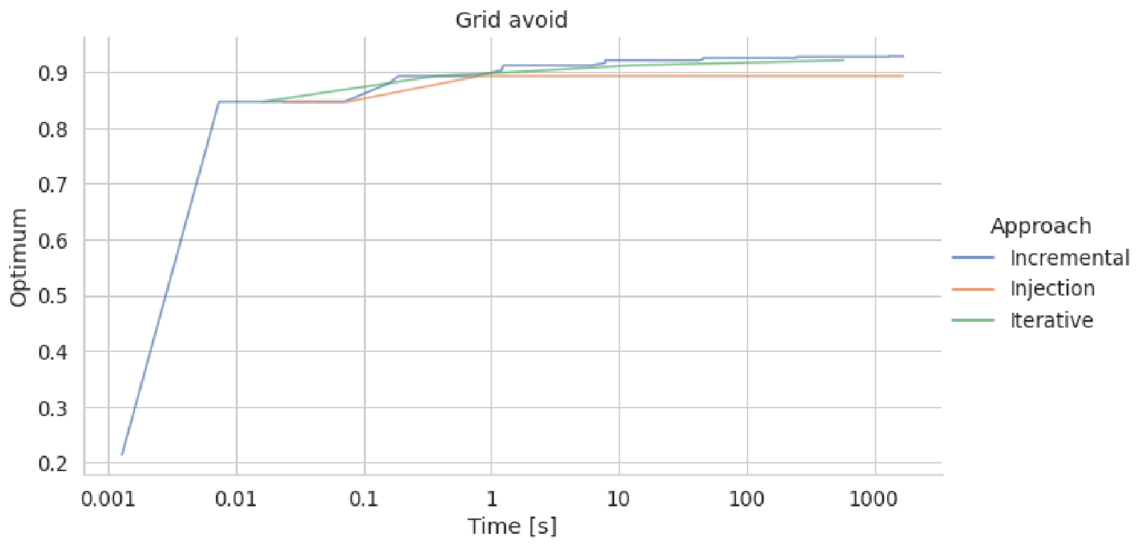


Figure 6.12: Graph comparing optimality value and time the Grid Avoid benchmark.

Had there been another restriction – design space – that PAYNT would need to explore, the incremental approach would likely not be able to find the controllers quicker than the iterative approach.

Grid Center

The Grid Center benchmark is similar to Grid Avoid both in design and in results. Figure 6.13 shows the results of each of the approaches – the optimality values and memory sizes. Figure 6.14 showcases the relation between elapsed time and best achieved optimums.

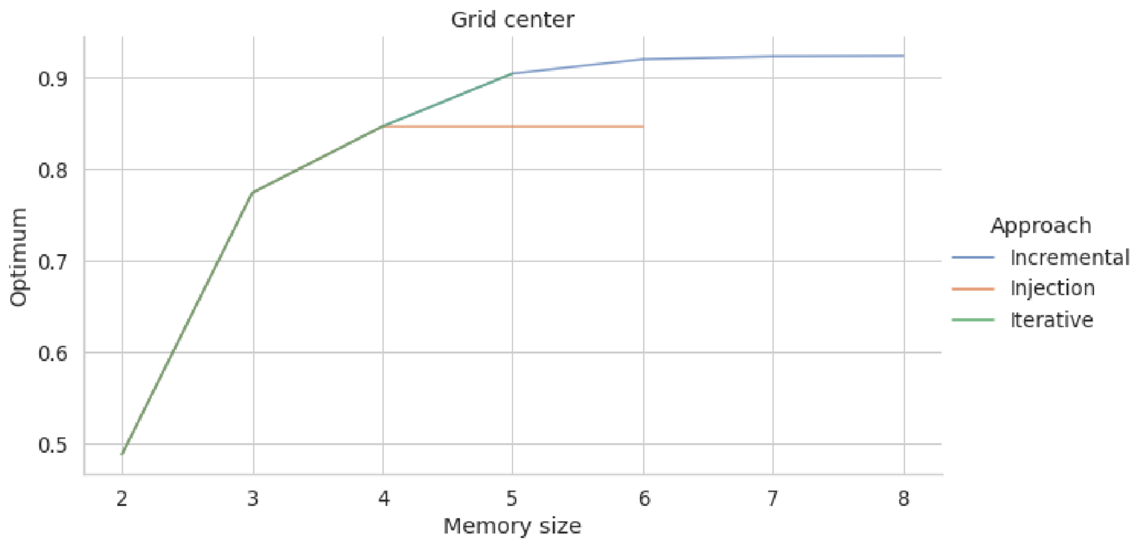


Figure 6.13: Graph comparing memory size and optimality value on the Grid Center benchmark.

Just like in the Grid Avoid benchmark, the iterative and incremental approaches are able to find controllers of the same quality in design spaces with memory sizes up to 5. Then, however, the iterative approach runs out of time while the incremental approach is able to continue and synthesize FSCs with memory sizes 6, 7, and 8 which each have a better optimality value than the previous one.

In this benchmark, the memory injection approach is able to find FSCs with the same quality as those found using the iterative and incremental approaches with memory sizes 2, 3, and 4. After that, it is not able to improve the controllers.

Compared to the Grid Avoid benchmark, in Grid Center the controller quality improves slower and more gradually over time in all three approaches.

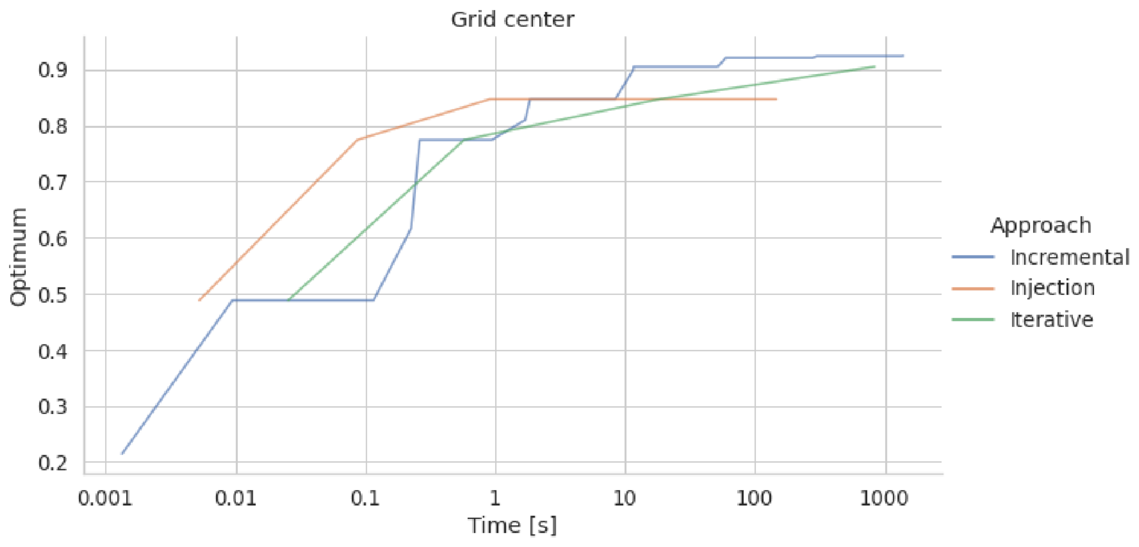


Figure 6.14: Graph comparing optimality value and time on the Grid Center benchmark.

6.3 Experiment Conclusions

Because this thesis only deals with preliminary experiments on the matter of design space restrictions, it is no surprise the results are mixed – in some cases, the experiment outcomes were quite unremarkable, in other cases the results look promising.

The least remarkable results were in the Maze benchmark. None of the tested approaches could synthesize a controller with better optimality value than 7.372105 in the given time regardless of memory size or the used restriction. However, given that the shortest possible path between the fields most distant from the target field is 7, the optimality value 7.372105 likely cannot be improved upon.

The Maze Long benchmark, where the shortest path from fields most distant from the target was 10 and the best found controller had the optimality value of ≈ 18 suggests that there is room for improvement. In this example, the controllers indeed seem to be improving in quality with added memory.

In the Grid Avoid and Grid Center examples, the results show an improving tendency in FSCs the larger the memory size is. Simple Circle seems to be a suitable restriction than can help finding these controller quicker.

Overall design spaces with larger memory sized appear to result in better FSCs in examples where there are multiple states with imperfect observations (e.g. observation 1 in Grid, observations 4 and 2 in Maze Long) which require the agent to take different actions. An imperfect observation by itself does not imply the need for the controller to have a bigger memory size. If in the system there is only one action the agent should take upon receiving the imperfect observation, there is no need to try to distinguish these states using memory values.

Chapter 7

Conclusion

In this thesis, I explored systems with state uncertainty modeled using partially observable Markov decision processes, what are their properties and how can an agent interact with them. Out of the two discussed methods for controlling agents with regards to POMDPs, the focus was mainly on finite-state controllers and their synthesis.

PAYNT is a tool which can find the optimal FSC in a design space that encompasses all possible FSCs of a certain size for the given POMDP. However, with increasing size of the design space, the synthesis time also increases and soon the results are unobtainable in a rational time.

Preliminary experiments on restrictions have shown that they are able to cut down the time necessary for finding the optimal finite-state controller. Whether the controller's quality was affected, however, depended on the restriction and the benchmark. E.g. the Simple circle restriction was ideal for the Grid Avoid benchmark, while the Forward restriction was not.

The incremental memory setting approach was also evaluated on the four benchmarks and it has proved to be comparable to the iterative and memory injection methods in some benchmarks (Grid Avoid, Grid Center) with regards to the quality of controllers found in the total elapsed time. A different set of restrictions could likely achieve better results.

Because larger memory size can vastly improve the quality of the controllers for POMDP which require different action in states with the same observations, future work related to this topic could explore different restriction or smarter ways of applying them in the incremental memory setting approach.

Bibliography

[1] Andriushchenko, R., Ceska, M., Junges, S. and Katoen, J.-P. Inductive Synthesis of Finite-State Controllers for POMDPs. 2022. Available at: https://arxiv.org/pdf/2203.10803.pdf.

[2] Andriushchenko, R., Češka, M., Junges, S. and Katoen, J.-P. Inductive Synthesis for Probabilistic Programs Reaches New Horizons. In: Groote, J. F. and Larsen, K. G., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Springer International Publishing, 2021, p. 191–209. ISBN 978-3-030-72016-2.

[3] Andriushchenko, R., Češka, M., Junges, S., Katoen, J.-P. and Stupinský Šimon. PAYNT: A Tool for Inductive Synthesis of Probabilistic Programs. In: *Computer Aided Verification. CAV 2021. Lecture Notes in Computer Science* [online]. Silva A., Leino K.R.M. (eds). Springer, Cham, 2021. ISBN 978-3-030-81685-8. Available at: https://link.springer.com/chapter/10.1007/978-3-030-81685-8_40.

[4] Baier, C. and Katoen, J.-P. *Principles of model checking*. MIT press, 2008.

[5] Češka, M., Hensel, C., Junges, S. and Katoen, J.-P. Counterexample-Driven Synthesis for Probabilistic Program Sketches. In: Beek, M. H. ter, McIver, A. and Oliveira, J. N., ed. *Formal Methods – The Next 30 Years*. Cham: Springer International Publishing, 2019, p. 101–120. ISBN 978-3-030-30942-8.

[6] Dehnert, C., Junges, S., Katoen, J.-P. and Volk, M. A Storm is Coming: A Modern Probabilistic Model Checker. In: Majumdar, R. and Kunčak, V., ed. *Computer Aided Verification*. Cham: Springer International Publishing, 2017, p. 592–600. ISBN 978-3-319-63390-9.

[7] *Graphviz* [online]. [cit. 2022-05-10]. Available at: https://graphviz.org/.

[8] *PyGraphviz* [online]. [cit. 2022-05-10]. Available at: https://pygraphviz.github.io/.

[9] Junges, S., Jansen, N., Wimmer, R., Quatmann, T., Leonore Winterer, J.-P. K. et al. Finite-state Controllers of POMDPs via Parameter Synthesis. In: *UAI 2018*. 2018.

[10] Knuth, D. and Yao, A. *Algorithms and Complexity: New Directions and Recent Results*. Academic Press, 1976.

[11] Kochenderfer, M., Wheeler, T. and Wray, K. *Algorithms for Decision Making*. 1st ed. MIT Press, 2022. ISBN 9780262047012. Available at: https://algorithmsbook.com.

[12] Kumar, A. and Zilberstein, S. History-based controller design and optimization for partially observable MDPs. In: *Proceedings of the International Conference on Automated Planning and Scheduling.* 2015, vol. 25, no. 1.

[13] Kwiatkowska, M., Norman, G. and Parker, D. *Stochastic Model Checking.* 2007. Available at: https://www.prismmodelchecker.org/papers/sfm07.pdf.

[14] Wray, K. H. and Czuprynski, K. Scalable POMDP Decision-Making Using Circulant Controllers. In: *ICRA 2021.* 2021.

[15] Češka, M., Jansen, N., Junges, S. and Katoen, J.-P. Shepherding Hordes of Markov Chains. In: Vojnar, T. and Zhang, L., ed. *Tools and Algorithms for the Construction and Analysis of Systems.* Cham: Springer International Publishing, 2019, p. 172–190. ISBN 978-3-030-17465-1.

Appendix A

Graphical FSC Representation

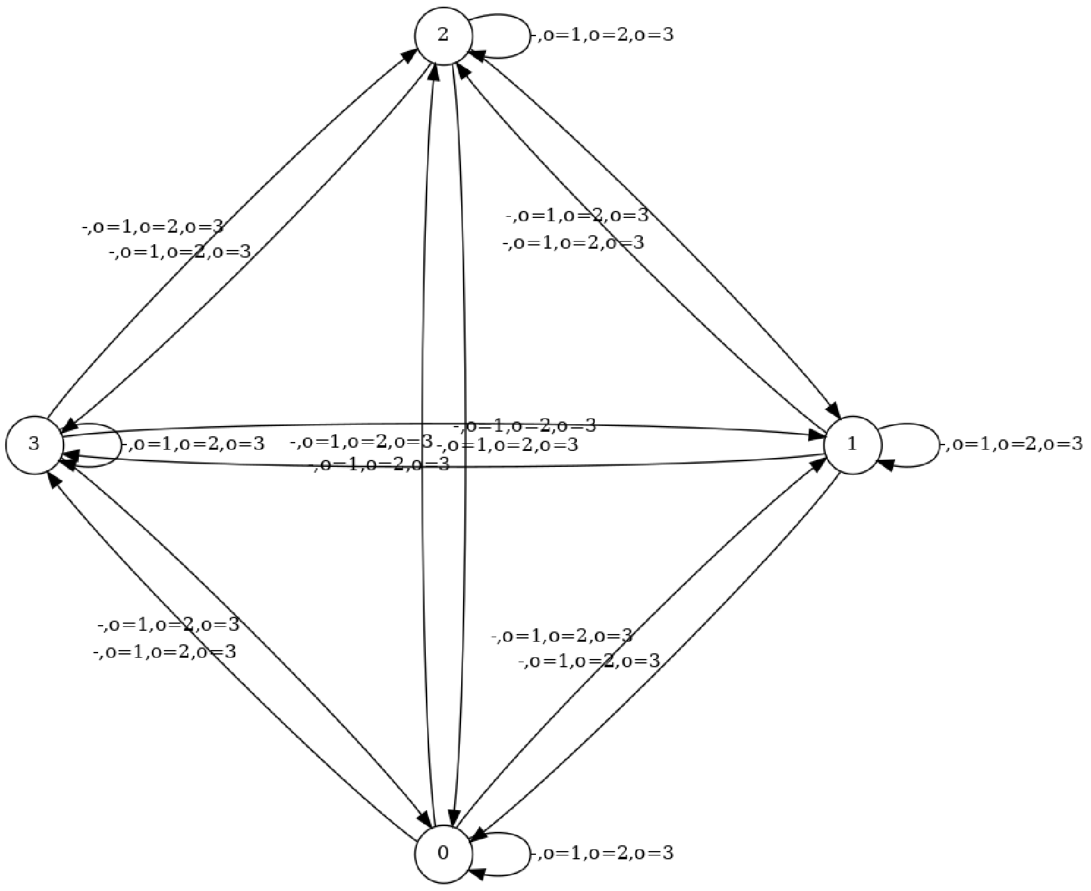


Figure A.1: Output of non-restricted design space of memory size 4, with observations

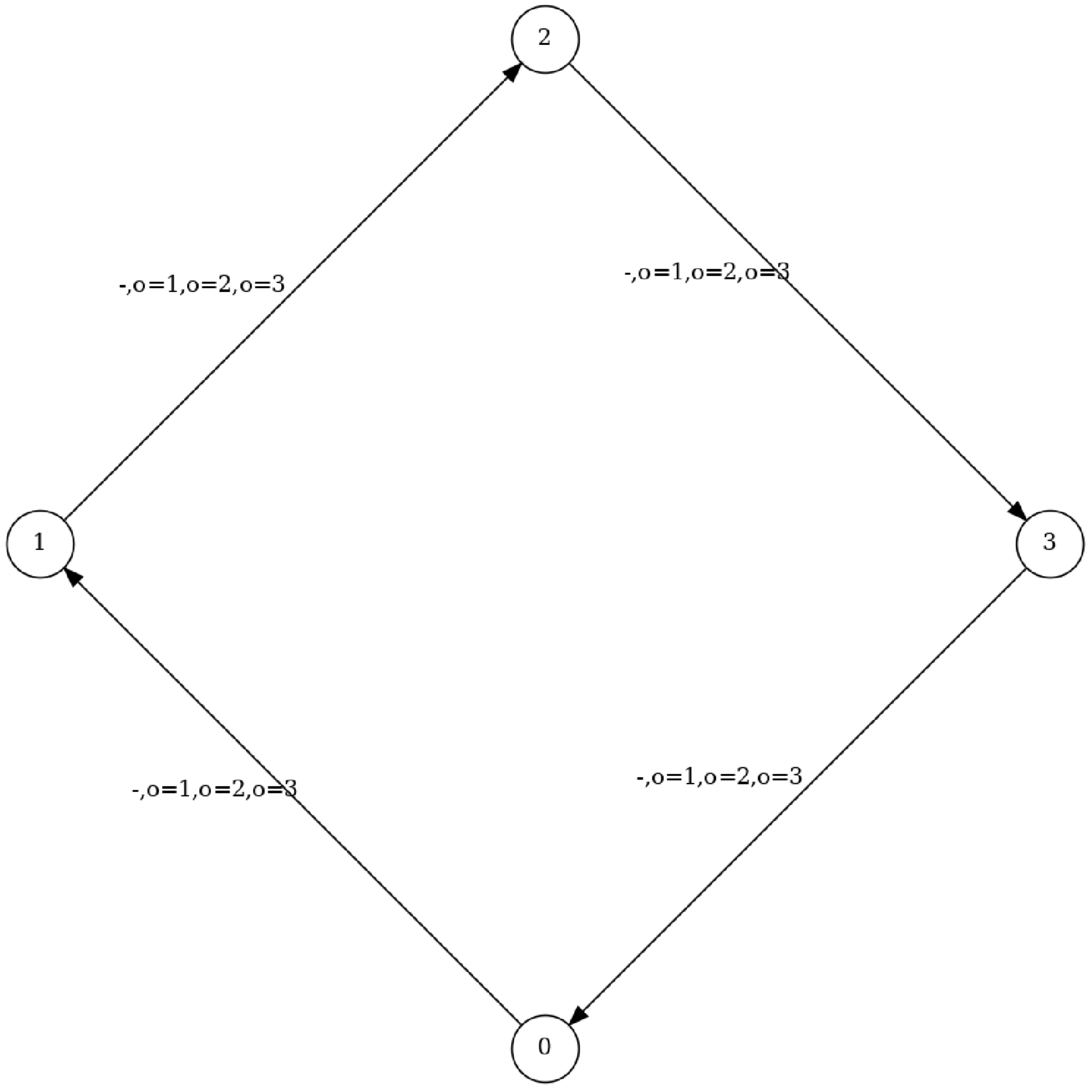


Figure A.2: Output of design space of memory size 4 restricted with Simple Circle, with observations

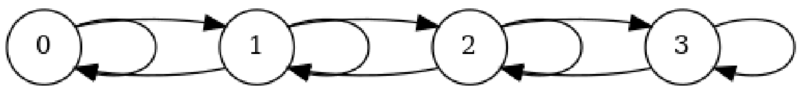


Figure A.3: Output of design space of memory size 4 restricted with One step, without observations

Appendix B

Contents of the included storage media

The included storage media contains the following files:

```
.
├── synthesis.zip
├── xgysel00.pdf
├── xgysel00_print.pdf
└── xgysel00.zip
```

Appendix C

Manual

PAYNT is tool that can automatically synthesize finite-state controllers for POMDPs. This fork modifies PAYNT to be able to restrict design spaces and only synthesize controllers of certain types.

Installation

To install the program, first unzip the `synthesis.zip` file and then from the `synthesis` folder run the installation script:

```
./install.sh
```

Running PAYNT

To run the program you use the script `./scripts/run.sh` which runs all three of the approaches on the four interesting benchmarks.

Before you run the program, make sure the folder `workspace/log/` exists and that you have the Python environment loaded (`source env/bin/activate`).

Options

```
Options:
  --project TEXT                  root  [required]
  --sketch TEXT                   name of the sketch file
  --properties TEXT               name of the properties file
  --fsc-synthesis                 enable incremental synthesis of FSCs for
                                  a POMDP
  --pomdp-memory-size INTEGER     implicit memory size for POMDP FSCs
  --incremental INTEGER...        enable incremental synthesis of FSC for
                                  a POMDP within a memory size with applied
                                  restrictions
  --strategy [full|iterative|injection]
                                  define strategy
  --reset-optimum                 reset the optimality property after each
                                  synthesis loop
  --help                          Show this message and exit.
```