Bachelor thesis
Computing Science

Radboud University

Connecting Mixed-Integer Linear Programming and Finite-memory POMDP Strategies

Author:
Tom Smitjes
s4599829

Supervisor/assessor:
dr. N.H. (Nils) Jansen
N.Jansen@cs.ru.nl

Second assessor:
dr. P.M. (Peter) Achten
P.Achten@cs.ru.nl

August 21, 2020

Abstract

We often make choices based on little information. An investment, quickest route home, one choice always turned out to be the best one. However, we want to know what the best choice is when we need to make it. Is there a way to predict which choice that will be?

In this thesis we will try to better understand *strategies* for dealing with a choices based on little and changing information. Based on the works of *Strengthening Deterministic Policies for POMDPs* ([17]), the objective of this thesis is to (a) understand the relation between *Partially Observable Markov Decision Process* (POMDPs) and *Mixed-Integer Linear Programming* (MILPs) and (b) attempt to create an MILP encoding which allows for POMDP strategies to automatically be generated.

Contents

Chapter 1

Introduction

1.1 A Robot's Journey

In order to get familiar with this topic, we will first take a look at a simple example. This will help us understand the research better.

1.1.1 Finding The Shortest Path (example)

Imagine: You and your robot have just landed on the planet *Examplarium*. After a day of exploring the robot needs to head back to the shuttle. It has to find its way back based on the map it has seen (seen in figure 1.1).

What is the minimum amount of actions the robot must make to go back to the shuttle?

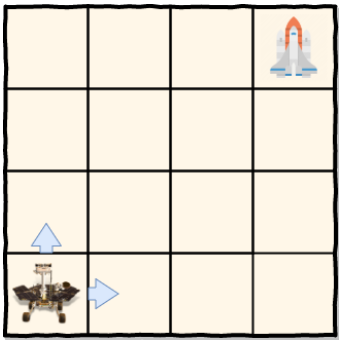


Figure 1.1: A map of *Examplairum*

As is visible from the map (1.1) the robot is currently on the bottom-left section of the planet. It needs to go to the top-right section of the planet to rejoin the shuttle. Every turn the robot can take an action to move to any adjacent section[1] or stay where it is.

Right now the robot has many different ways of getting home. It can move 3 times up followed by 3 times right ($start \xrightarrow{up} \xrightarrow{up} \xrightarrow{up} \xrightarrow{right} \xrightarrow{right} \xrightarrow{right} finish$) or any other combination of going 3 times up and right. Going left or down would only mean that the robot would have to go right or up again and so that would be inefficient. Similarly, staying in the same section will only cause delay. Therefore the quickest route would need **6 actions**.

[1] adjacent = up, down, left, right

1.1.2 Finding The Best Strategy (example)

But then: *The robot receives an alarming signal from the shuttle. A dangerous storm has come up! It needs to head back as soon as possible!*

The robot is send the current location of the storm and knows the probabilities of the storm's movement. However, it can only see the sections adjacent to it, and so can't be sure of the storms location after it has moved.

The robot has to go back to the shuttle as soon as possible while also avoiding the moving storm. What is the best ***strategy?***

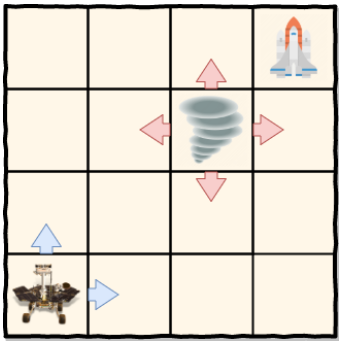


Figure 1.2: A storm has come in Examplairum

As we can see from the map (1.3) the robot and the shuttle are positioned like before, but there is now also a randomly moving storm. The shuttle has been made storm proof, but should the storm and the robot end up in the same tile then the robot will immediately be destroyed.

The storms position is only given at the beginning. After each turn the storm can move to an adjacent section[2] or stay. Each of its moves has an equal probability *(20% up, 20% down, 20% left, 20% right, 20% stay)*. If the storm were to move off the map it will stay in its current position.

The storm moves first, followed by the action of robot, then the storm again, etc.. How can we find the best actions for the robot to go back as quickly as possible while also avoiding the storm as best as possible?

Understanding This Problem

To be able to visualize what's going on we will use *different versions of the map*, each with their own probability. After the storm moved, when it can be in one of five places, we will create a total of five different map-states. Each of these states represents a different storm-location, with each map-state having their own probability of the storm being there[3].

[2] adjacent = up, down, left, right

[3] In the beginning that means five map-states, one for each of the five storm movements, with 20% chance associated to each of these map-states

Whenever the robot takes an action, we will make it take that action on every map-state. If on any one of these map-states the robot were to collide with the storm, we will stop using that state and add that states probability to an overall *chance that the robot won't make it.*

After each of the robots actions, for each of all the currently used map-states, we will create new map-states based on all possible moves the storm could take from that state. We will then add all of these to the new set of map-states to be used for the next robot action.

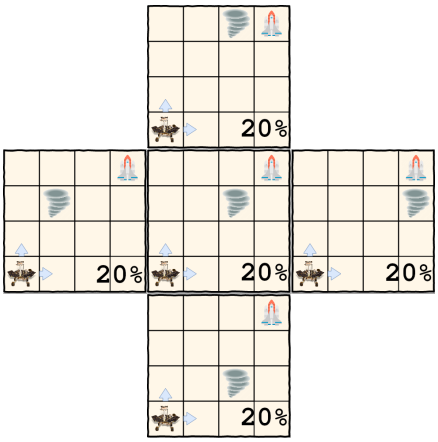


Figure 1.3: Map-states of the scenario after one turn of the storm

Luckily the robot is not powerless: It can see an oncoming storm if it is in *any of its adjacent tiles.* Once we have seen the storm we know where the storm is with a 100% certainty, and so we can discard all other map-states.

Solving This Problem

When the robot sees the storm we can right then change its path accordingly. For example, we can instruct that the robot must take the actions as before as much as it can, but must try to dodge the storm if it sees it, whenever this will be. This set of instructions which allow us to spontaneously change our path we will call a *strategy.*

Simple strategy

Try to move like we did ($start \xrightarrow{up} \xrightarrow{up} \xrightarrow{up} \xrightarrow{right} \xrightarrow{right} \xrightarrow{right} finish$) but if we see the storm blocking the route, we will try to move *backwards* (down or left). If not, continue.

Because we can change our path according to what we see, instead of one optimal *path* we are looking for one optimal *strategy.* A strategy which is able to offer *new paths with new information*[4]. To understand more about the effectiveness of different strategies we will first need to understand this problem better from a theoretical viewpoint[5].

[4]In this case, seeing the storm alters the path

[5]More about this example and possible strategies in the appendix

1.2 Finding A Strategy

With our robot we wanted to go back to the shuttle while also avoiding the storm. For this we needed to find the *optimal strategy* for that problem. To be able to understand such problems better, we will use *Partially Observable Markov Decision Processes* (POMDPs). We will then try to relate our problem to algorithms using *Mixed Integer Linear Programming* (MILP).

> **POMPDs**[6]. A POMDP[12] is a framework where in we can explain problems like our example with actions, states, paths and strategies. Just like how we can solve other problems by thinking of it mathematically, a POMDP provide a framework which can help us to find the answer to such problems.

> **MILPs**[7]. To be able to find an optimal POMDP strategy, we will use MILPs[17]. A MILP is a program structured in a very specific way, using only a goal, constraints and variables. We want to use MILPs because they are *solved*, meaning that we can automatically (via a program) find the answer to any mathematical problem as long as we can structure it as a MILP.

1.2.1 Research Question

To solve such problems better, the research question of this thesis is:

How does Mixed-Integer Linear Programming relate to the computation of finite-memory strategies for a POMDP?

We want to find the optimal POMDP strategy automatically. Because we know that MILPs are *solved*, should we be able to find such a mathematical relationship between any POMDP and a matching MILP, we will be able to create optimal POMDP strategies automatically. This is why finding this relationship is so important.

[6] More on the definition of a POMDP in the preliminaries.
[7] More on the definition of a MILP in the preliminaries.

1.3 Motivation

Apart from our example in the introduction (1.1.2) understanding POMDPs and learning how to work with them has been a much covered area of research. Similarly, the need for a better way to work with POMDPs has been growing with the growth of the popularity of POMDPs in practical environments.

1.3.1 Own Examples

We can see the potential use of POMDPs everywhere. Whether we are estimating a shortest route, guessing intentions of another person or otherwise making decisions when we don't know the full picture. Applying the right POMDP strategy can help solve a lot of potential issues.

Real Option Valuation

One of the more lucrative areas where we could see such use is in Real option valuation. This is a process by which an investor estimates whether an investment is worthwhile or whether it is better to abandon it.

Other Examples

Other areas where we may see POMDPs include games (where we do not know what the roll of the die will bring), advertisement (where we do not know how many people will watch our adds) or strategical decisions (where we do not know of the enemies intentions). All these scenarios are scenarios where we need to make choices based on estimates, and it are these estimates which we can learn to understand using a POMDP.

1.3.2 Related Work

POMDPs are not a new area of research. As shown by papers such as those who cover multiple POMDP related issues [9] it has been wildly covered, and this is not without a reason. Because of the way POMDPs are build, they cover a potentially endless amounts of scenarios, ranging from speech recognition [8], dealing with demenatia [7], keeping track of near extinct species [6] and even keeping track of inmates [15].

However, actually solving a POMDP problem once it has been described as such still seems to be a challenge. In the words of Matthijs T.J. Spaan, *"As solving POMDPs optimally is very hard, we will have to consider approximate algorithms."* ([15], page 51). Though useful as thinking of problems as a POMDP may appear, it helps nothing if there are no algorithms which can find solutions easily.

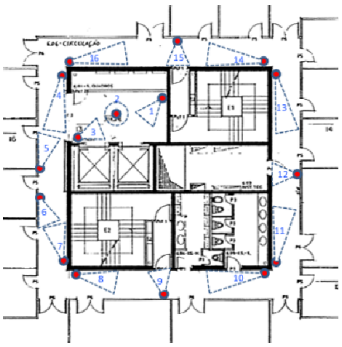


Figure 1.4: Map of the 6th floor of ISR, Lisbon, Portugal[15]. Even for prison cells can POMDPs be useful

1.3.3 Our Solution

Should we be able to relate any POMDP to a MILP and so find proper algorithms that we can use to find the solution to any POMDP problem, we will have, by extension, provided a solution to many different problems at the same time.

1.4 Overview

We will first cover the basic definitions needed to understand this paper in the *preliminaries*. We will then, in *research* chapter, go over two interpretations of what an optimal strategy could be, and we will try to create two MILP algorithms which find matching optimal strategies. We will then conclude with a working implementation in python code. We will then go over this code and discuss what further research can still be done.

Chapter 2

Preliminaries

To understand the research we will need to understand the topics and definitions. In this thesis we will mostly be using the definitions as defined in *Permissive Finite State Controller Of POMDPs* [12] and *Strengthening Deterministic Policies for POMDPs* [17].

2.1 Distributions

In this thesis we will use *distributions*. With distributions, instead of returning just one element from a set, we can talk about *possible outcomes from a set*, each with their own *probability*.

Definition 1 (Distribution)**.** *For a set X with elements e_1, e_2, ... , e_n, a distribution of that set is defined as:*

$$(p_1 : e_1,\ p_2 : e_2,\ ....\ ,\ p_n : e_n) \in Distr(X)$$

with every p_i the probability of element e_i, with $0 \leq p_i \leq 1$, and with the sum of the probabilities in a distribution being one ($\sum_{e_i \in X} p_i = 1$).

For example we can create the distribution of a fair die, with each side of the die having a one in six probability of being rolled, as:

$$(\frac{1}{6} : one,\ \frac{1}{6} : two,\ \frac{1}{6} : three,\ \frac{1}{6} : four,\ \frac{1}{6} : five,\ \frac{1}{6} : six)$$

If the probability of an element is zero, we won't write them down. For example, an unfair die that always rolls a six can be written as $(1 : six)$.

As A Function

We can use a distribution $distr \in Distr(X)$ as a function, which for any element $e \in X$ will return the probability p of that element in $distr$. This would look like $distr(e) = p$.

2.2 Definition POMDP

Partially Observable Markov Decision Processes (POMDPs) are derived from *Markov Decision Processes* (MDPs)[12]. To understand POMDPs we will first need to understand MDPs.

2.2.1 MDPs

In the *introduction* (1.1.1) we came across an example where after performing a certain *action* from a certain *state*, we end up in a random new *state*. We could never choose in which state we would end up, only our actions.

A *Markov Decision Process* (MDP) is a mathematical framework which can be used to describe a scenario like our example. It consists of a set of all states (S), a state from which we begin (s_i), a set of all actions (Act), and a probabilistic function (P) which, after performing an action from a certain state, will give us a distribution of where we will end up.

Definition 2 (Markov Decision Process (MDP))**.**
A MDP is defined as a tuple $\mathbb{M} = (\boldsymbol{S, s_i, Act, P})$*, where:*

- $\boldsymbol{S}$ *- A finite set of all possible states.*
- $\boldsymbol{s_i}$ *- The initial state.*
- $\boldsymbol{Act}$ *- A finite set of all possible actions.*
- $\boldsymbol{P}$ *- The probabilistic function* $P : S \times Act_s \to Distr(S)$*. This gives a distribution of all states that you might end up in after performing a certain action from a certain state.*

$Act_s \subseteq Act$ is the set of all actions which are possible from a state s, because it is not always is every action possible[1]. We will also define $succ(s, a) \in S$ as the set of states reachable from state s using action a.

MDP Example

In the example (A.1)[2] we describe a coin-toss using a MDP. We use the P function to describe that we can end up on any of the two sides after a toss. The MDP looks as follows:

$$\mathbb{M}_c = (S_c, s_{i\,c}, Act_c, P_c)$$

with $S_c = \{heads, tails\}$, $s_{i\,c} = heads$, $Act_c = \{toss\}$ *and*
$P_c(heads, toss) = P_c(tails, toss) = (0.5 : heads, 0.5 : tails)$

[1] In the introduction we could not move further down once we were at the bottom.
[2] See the appendix for a more in-depth explanation

2.2.2 POMDPs

In the second robot example (1.1.2) we often did not know in which map-state the robot was. Instead, we could only *guess the current state based on limited knowledge* (like where the storm started, the amount of turns, etc.). We will call all current knowledge of our state our *observation*.

To show this uncertainty we will add *observations*. We will extend MDPs to *Partially Observable Markov Decision Processes* (POMDPs) with the addition of two parameters, a set of observations (Z) and a mapping function (O) which matches every state to its observation.

Definition 3 (Partially Observable Markov Decision Processes (POMDP))**.** *A POMDP is defined as a tuple* $\mathbb{P} = (\boldsymbol{M}, \boldsymbol{Z}, \boldsymbol{O})$*, where:*

- $\boldsymbol{M}$ *- A MDP.*
- $\boldsymbol{Z}$ *- A finite set of observations.*
- $\boldsymbol{O}$ *- The observation function* $O : S \to Z$*. This function maps a state to the corresponding observation.*

We will also define $z_i = O(s_i)$ as our initial observation and $states(z) = \{s \mid O(s) = z\}$ as the set of possible states given an observation z.

POMDP Belief

In a POMDP, for an observation z, all states in $states(z)$ need to have the *exact same actions* (Act_z). However, the outcomes of these actions *can* vary. To estimate in which state we are we can use *belief*.

Definition 4 (Belief)**.** *For* z *the current observation, the belief* $b \in Distr(states(z))$ *is the distribution of possible current states.*

For example, when we perform an action from the state s_i, we could end up in a number of states based on the P function. If we then eliminate all the states that do not match our current observation, we get our belief[3].

POMDP Example

In the appendix (A.2)[4] we extend our MDP to a POMDP, as follows:

$$\mathbb{P}_c = (M_c, Z_c, O_c)$$

with $M_c = \mathbb{M}_c$, $Z_c = \{mystery\text{-}side\}$ *and*
$O_c(heads) = O_c(tails) = mystery\text{-}side$

[3]Similarly we can do the same thing with every following action, only then we do it for each of the possible states and then add the probabilities appropriately.

[4]See the appendix for a more in-depth explanation.

2.3 Definition Path

In the first part of introduction we were moving the robot. These moves that the robot made is our *path* .

Definition 5 (POMDP paths)**.** *A path $\pi \in$ Paths in a POMDP $\mathbb{P}$ is defined as the sequence of states and actions:*

$$\pi = s_i \xrightarrow{a_1} s_2 \xrightarrow{a_2} \dots \xrightarrow{a_{n-1}} s_n$$

Here each $s_x \in S$ is a state with s_i the initial state, and each $a_x \in Act_{s_x}$ is the action taken after that observation.

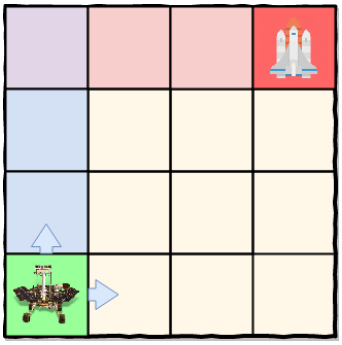


Figure 2.1: Examplairum with the path colored

We can think of a path as a series of actions and states we would end up. For example, in the first part of the introduction we continuously made the action of moving upwards or right. We described our path as:

$$start \xrightarrow{up} \xrightarrow{up} \xrightarrow{up} \xrightarrow{right} \xrightarrow{right} \xrightarrow{right} finish$$

For a POMDP path we also remember the *state* after each action. Every time the robot moved the map changed to show the robots movement. This new map is our new *state*. With this our POMDP path would look like:

$$\pi_{example} = s_i \xrightarrow{up} s_2 \xrightarrow{up} s_3 \xrightarrow{up} s_4 \xrightarrow{right} s_5 \xrightarrow{right} s_6 \xrightarrow{right} s_7$$

All paths begin with s_i and end with a state we call $last(\pi)$ (In our example, $last(\pi_{example}) = s_7$). From that last state we can usually perform a new action and end up with a new observation. To remember this action and observation we simply add them to the end of the previous path:

For a previous path $\pi_{\lambda prev} = \dots s_n$, after performing action a_n resulting in state s_{n+1}, the next path is: $\pi_{next} = \pi_{prev} \xrightarrow{a_n} s_{n+1} = \dots s_n \xrightarrow{a_n} s_{n+1}$

With this we can use paths to *"walk"* through a POMDP, with the path showing us the history of our states and actions. Walking through a POMDP step-by-step is what we will use for our strategies.

2.3.1 Observation Path

In a POMDP we only know of our observations. Because of this we will define $\theta = O(\pi) \in Paths$ to describe a path π for which all states are replaced with their matching observations by the O function.

2.4 Definition Strategy

In the second part of the introduction we saw a randomly moving storm. It was clear that trying the same path could result in our robot being destroyed, and so we needed a *strategy*.

Definition 6 (POMDP strategy)**.** *A strategy for a POMDP* $\mathbb{P}$ *is defined as the function:*

$$\sigma : Paths' \to Distr(Act_Z)$$

This function takes our current path $\theta \in Paths'$ *and returns a distribution of actions from* $Act_{last(\theta)}$ *that we should take.*

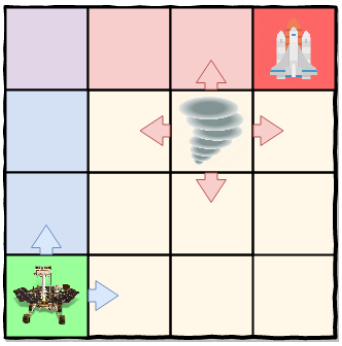


Figure 2.2: Examplairum with the path and storm

This strategy function can be set and changed. We can freely choose what actions $\sigma(\theta)$ returns. What we choose will tell a user of our strategy what actions we think they should take. To demonstrate this, we can try to rewrite our *simple strategy* from the introduction:

Simple strategy

Try to move like we did ($start \xrightarrow{up} \xrightarrow{up} \xrightarrow{up} \xrightarrow{right} \xrightarrow{right} \xrightarrow{right} finish$) but if we see the storm blocking the route, we will try to move *backwards* (down or left). If not, continue.

With a little help from the coloring from figure 2.2 we can set our σ function such that its results are the same as our *simple strategy*[5]:

Simple strategy - POMDP version

For any path $\theta \in Paths'$:
- If *the storm is not blocking the path*:
  - If $last(\theta)$ is in the *green* or *blue* area: $\sigma(\theta) = (1 : up)$
  - Else if $last(\theta)$ is in the *purple* or *red* area: $\sigma(\theta) = (1 : right)$
- Else if *the storm is blocking the path*:
  - If $last(\theta)$ is in the *blue* or *purple* area: $\sigma(\theta) = (1 : down)$
  - Else if $last(\theta)$ is in the *red* area: $\sigma(\theta) = (1 : left)$
  - Else if $last(\theta)$ is in the *green* area: $\sigma(\theta) = (1 : stay)$

As we can see in the POMDP version of our strategy, for any path, depended on whether there is a storm blocking the path and where it is now, we get an action that we must take[6], similar to our *simple strategy*.

[5] A few simplifications have been made to make the algorithm more understandable.

[6] In this example the strategy only recommend *one* action each time with a 100% certainty. This is called a *deterministic* strategy.

2.4.1 Finite-memory Strategy

Paths can be infinitely long, and for each of these paths our strategy can recommended something unique. However we are trying to automatically generate a strategy, and no computer can store an infinite memory.

We need to limit ourselves to a *finite-memory* strategy, one that remembers only so much of our current path. Because of this we will define the function $last_m$, which strips any path to its last m states and in-between-actions. For example:

$$last_3(s_4 \xrightarrow{a_4} s_5 \xrightarrow{a_5} s_6 \xrightarrow{a_6} s_7 \xrightarrow{a_7} s_8) = s_6 \xrightarrow{a_6} s_7 \xrightarrow{a_7} s_8$$

We will also define $Paths_m$ as *the set of all possible paths (not necessarily starting at s_i) of at most length m.* We will use both $Paths_m$ and $last_m$ to create the definition of finite-memory strategies.

Definition 7 (Finite-memory POMDP Strategy)**.** *A finite-memory strategy for a POMDP $\mathbb{P}$ is a strategy for a finite path $\theta \in Paths'_m$:*

$$\sigma : Paths'_m \to Distr(Act_Z)$$

with for every path $\theta \in Paths'$ the strategy is equal to $\sigma(last_m(\theta))$.

Using this kind of strategy we only need an output for a finite amount of inputs. This we are able to model in a computer.

2.5 Definition MILP

To be able to automatically create a strategy for any POMDP, we will be using *Mixed Integer Linear Programming* (MILP). This mathematical optimization program allows us to optimize a number of *variables* according to a *goal* and its *constraints*. MILPs are *solved*, meaning that we are able to optimize any MILP with the help of a computer.

Definition 8 (Mixed Integer Linear Program (MILP))**.** *A MILP is an optimization program with three parts:*

- **Goal** - *An expression that needs to be **maximized** or **minimized***
- **Constraints** - *A set of linear equations that must be true*
- **Variables** - *The variables that need to be optimized*

For example, we can create a MILP with variables x and y, goal "**maximize** x", and with the constraints $x < y$ and $y < 3$. The optimal solution of that MILP would be $y = 3$ and $x = 3$.

Chapter 3

Research

3.1 Research Approach

The research question of this thesis is:

How does Mixed-Integer Linear Programming relate to the computation of finite-memory strategies for a POMDP?

To answer this question we will try to automatically create POMDP strategies with MILP algorithms using *Strengthening Deterministic Policies for POMDPs* [17]. If we were to successfully do this, we will have shown that there exists integrate relation between the two.

3.1.1 Optimal Strategy

The definition of an "optimal" strategy is not well defined, and multiple scenarios may have multiple perspectives. Because of this we will create two different algorithms, each with their own perspective:

- **Success States.** We will design an algorithm based on *success and failure states*, which will have states which need to be *reached* (resulting in a "success"), and states which need to be *avoided* (resulting in a "failure"). This could be useful for calculating a shortest route.
- **Reward**. We will alter our algorithm based on *rewards*. This is a number which we will aim to increase as much as possible throughout the algorithm. This is may be useful for financial decisions.

Using these two different perspectives we will try to still cover all of what an "optimal" strategy could be.

3.2 Creating An Algorithm

Before we can implement any algorithm, we first need to make sure that we will be able to create such an algorithm in the first place.

3.2.1 Computable

To make sure that we are able to model the POMDP in a computer, it is important that all our data is *finite*[1]. This means that the set of states S, actions Act and observations Z are finite[2].

Modeling The Strategy

Our strategy also needs to be a *finite-memory strategy*. Such a strategy can be defined as a function which, from a path θ, returns a *distribution of actions*. Alternatively, we can define our strategy by each of these probabilities *individually* for each action, as $\sigma(\theta)(a)$, with $\forall\theta : \sum_{a \in Act_{last(\theta)}} \sigma(\theta)(a) = 1$. Using this definition we can better model our strategy.

3.2.2 (Non-)Deterministic Strategies

For the algorithm we will use *deterministic* strategies. These strategies only recommend *one* action from any path, with all other actions discarded. Deterministic strategies are easier to model in an algorithm.

Definition 9 (Deterministic strategy)**.** *A strategy σ is deterministic if, for all paths θ and possible actions a: $\sigma(\theta)(a) = 0$ or $\sigma(\theta)(a) = 1$.*

Non-deterministic strategies can recommend more than one action, each with a probability between zero and one[3]. This allows for recommending multiple actions with limited certainty.

Definition 10 (Non-deterministic Strategy)**.** *A strategy σ is non-deterministic if, for all paths θ and possible actions a: $0 \leq \sigma(\theta)(a) \leq 1$*

Ultimately, non-deterministic strategies cover a wider range of possibilities than deterministic strategies. This is why we will later try to convert our algorithms so that they also work for non-deterministic strategies.

[1] Finite does not mean small! Finitely big numbers include $10^{10^{10^{10}}}$, a google, etc.
[2] $|Z| \leq |S|$ unless there are unreachable observations, which doesn't make sense.
[3] The sum of all the probabilities for a given path must still be 1.

3.3 Strategy Algorithm (Success States)

The first approach to creating an algorithm will use *success and failure states*. Using this method for our algorithm, we are able to solve problems involving states that we want to reach versus states that we want to avoid[4].

3.3.1 Success And Failure States

This algorithm will try to optimize a strategy based on a non-empty set of *success states* (S_S) in S. We want to end up with these states, and so getting there as quickly as possible is one of the goals of the algorithm.

Definition 11 (Success States)**.** *A non-empty set of observations $S_S \subset S$ which result in an immediate **success**.*

However, we will also define the set of *failure states* (S_F) in S. This set *can be empty* and has *no overlap with* S_S. As soon as we are in one of the failure observations we will *fail*, even if more actions after that were possible. Minimizing the probability of ending up in one of these states is the other goal of our algorithm.

Definition 12 (Failure States)**.** *A set of observations $S_F \subset S$ which result in an immediate **failure**. $S_F \cap S_S = \emptyset$*

For example, in the introduction we wanted our robot to be back to the shuttle as quick as possible, so any states where the robot was at the shuttle was a *success states*. However, we also wanted to avoid the destructive storm as best as we could, and so any states where the robot and the storm would be in the same section was a *failure states*.

3.3.2 The p_s Function

To calculate the best strategy in a POMDP, we will make clever use of several "smaller than"-functions and one value which we need to maximize. This way the strategy will naturally favor the best actions. It will do so using the p_s function.

Definition 13 (p_s function)**.** *For a deterministic strategy, p_s is defined as the probability (between 0 and 1) of reaching a state in S_S from a state s according to our current strategy σ. p_s is defined using three equations[5][6]:*

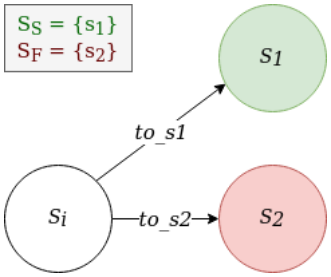
- $\forall s \in S_S : p_s = 1$ *(1)*
- $\forall s \in S_F : p_s = 0$ *(2)*
- $\forall s \in S^*, a \in Act_s : p_s \leq (1 - \sigma(\theta)(a)) + \sum_{s' \in succ(s,a)} P(s,a)(s') * p_{s'}$ *(3)*

[4] For example this can be used for winning a game, finding a moving target, etc.
[5] $succ(s,a)$ returns the next possible states after performing a from s.
[6] $S^* = S \setminus S_S \cup S_F$

Example p_s Function

To demonstrate how this p_s function works, we look at an example:

> *Say we have a POMDP with states* $\{s_i,\ s_1,\ s_2\}$,
> *actions* $\{to\_s1,\ to\_s2\}$ *and* P *such that* $P(s_i, to\_s1) = (1 : s_1)$ *and* $P(s_i, to\_s2) = (1 : s_2)$.
>
> *Say* $S_S = \{s_1\}$ *and* $S_F = \{s_2\}$. *With other words, we want to take action* to_s1 *from* s_i *as much as possible.*



> *It follows from (1) and (2) that* $p_{s_1} = 1$ *and* $p_{s_2} = 0$
>
> *Using (3), with* $Act_{s_i} = \{to\_s1,\ to\_s2\}$, $succ(s_i, to\_s1) = \{s_1\}$ *and* $succ(s_i, to\_s2) = \{s_2\}$, *we get the following equations:*

$$p_{s_i} \leq (1 - \sigma(z_i)(to\_s1)) + p_{s_1} = (1 - \sigma(z_i)(to\_s1)) + 1$$

$$p_{s_i} \leq (1 - \sigma(z_i)(to\_s2)) + p_{s_2} = (1 - \sigma(z_i)(to\_s2))$$

> *The goal of the algorithm will be to maximize* p_{s_i} *as much as possible. For this, either* $\sigma(z_i)(to\_s1) = 1$ *and* $\sigma(z_i)(to\_s2) = 0$, *or the other way around.*
>
> *- If* $\sigma(z_i)(to\_s1) = 1$ *then* $p_{s_i} \leq 1$ *and* $p_{s_i} \leq 1 \implies max\ p_{s_i} = 1$
> *- If* $\sigma(z_i)(to\_s2) = 1$ *then* $p_{s_i} \leq 2$ *and* $p_{s_i} \leq 0 \implies max\ p_{s_i} = 0$
>
> *It follows that we get the highest* p_{s_i} *with* $\sigma(z_i)(to\_s1) = 1$. *With other words, the strategy recommends taking action* to_s1, *which is what we wanted.*

Getting Stuck

It may happen that our strategy would recommend to take an action that would lead us directly to a state or group of states where we can't escape from. These states would not bring us any closer to our goal, but there are also no actions possible that could ever return us back on our path. This is what we will call "stuck".

Reachability Rank

To ensure that the strategy will not get stuck, we will use a *reachability ranking* r_s (between 0 and 1) which, using a similar method as the p_s function, will indicate how close we are to reaching our goal. The higher this value is, the better.

We will also define a variable $t_{(s,s')}$ which can be either 0 or 1, depending on whether we will move closer (one) or further away (zero) from our goal. This will act as a barrier, and will close of paths which will get us stuck.

We will define r_s and $t_{(s,s')}$ using the following three equations:

- $\forall s \in S^*,\ \textit{if } Act_s = \emptyset : r_s = 0$ *(1)*
- $\forall s \in S^*, s' \in succ(s) : r_s < (1 - t_{(s,s')}) + r_{s'}$ *(2)*
- $\forall s \in S^*, a \in Act_s : p_s \leq (1 - \sigma(\theta)(a)) + \sum_{s' \in succ(s,a)} t_{(s,s')}$ *(3)*

With these equations, any loop or dead-end will have $t_{(s,s')} = 0$ somewhere for two consecutive states[7]. This will make such paths undesirable when trying to maximize p_{s_i}, and so will make sure that actions which leads to getting stuck such as these will preferably not get chosen.

[7]This is because r_s always needs to be strictly smaller than $r_{s'}$. Should they loop, then there must be a point where $t_{(s,s')} = 0$, making actions leading to there less attractive.

3.3.3 MILP Encoding

Using the equations and theory as we described earlier, we will be able to create an algorithm that can optimize a strategy so that we try to *reach* certain states (S_S), but *avoid* other states (S_F).

Initiation

The algorithm will expect the following input:

- $\mathbb{P}$ - The POMDP for which we want to find an ideal strategy
- $\boldsymbol{S_S}$ **and** $\boldsymbol{S_F}$ - The success and failure states

Algorithm

Using the above input, we will use the following MILP encoding[8]:

Algorithm 1: MILP Success Strategy Creator

Data: $\mathbb{P}$, S_S, S_F

Goal: maximize p_{s_i} (maximize probability of ending in Z_S)

Constraints:

$\forall a, \theta : \sigma(\theta)(a) = 0$ or $\sigma(\theta)(a) = 1$ (deterministic strategy)

$\forall \theta : \sum_{a \in Act_{last(\theta)}} \sigma(\theta)(a) = 1$

$\forall s \in S : 0 \leq p_s \leq 1$

$\forall s \in S_S : p_s = 1$

$\forall s \in S_F : p_s = 0$

$\forall s \in S^*, a \in Act_s : p_s \leq (1 - \sigma(\theta)(a)) + \sum_{s \in succ(s,a)} P(s,a)(s') * p_{s'}$

$\forall s \in S : 0 \leq r_s \leq 1$

$\forall s \in S, s' \in succ(s) : t_{(s,s')} = 0$ or $t_{(s,s')} = 1$

$\forall s \in S^*, \; if \; Act_s = \emptyset : \; r_s = 0$

$\forall s \in S^*, s' \in succ(s) : r_s < (1 - t_{(s,s')}) + r_{s'}$

$\forall s \in S^*, a \in Act_s : p_s \leq (1 - \sigma(\theta)(a)) + \sum_{s' \in succ(s,a)} t_{(s,s')}$

Variables: $\sigma(\theta)(a)$, p_s, r_s, $t_{(s,s')}$

In the algorithm we use the p_{s_i} function as described earlier. Using that function we can get the probability of ending up in a state that matches on of the success states. We also wont want to take any step towards states that will lead to dead-ends or lead us to failure states.

[8] $S^* = S \setminus S_F \cup S_S$

3.4 Strategy Algorithm (Reward)

With our previous algorithm we tried to create an algorithm based on *success states*. Now we will try to create our strategy based on *rewards*. A reward is a value which we want to increase as much as possible during the algorithm.

3.4.1 Reward

We will give every state and the action that we took from that state a value. This value will represent the *reward* that that interaction will give us. We will use the function $r(s, a)$[9] for s a state and a an action in A_s.

Current Reward

Each time we perform a new action from a new state, we add the matching reward to our *current reward* (v_s). We will keep adding new rewards until we stop. This way we can better simulate the idea of finding profitable routes which will increase our reward.

Reward Decay

It may happen that the algorithm discovers a loop where we will gain more reward with every cycle. Would that be the case we could go around endlessly, ever increasing our reward till infinity. Such a loophole is of course not desirable or realistic.

To make sure that such loops wont interfere, we will alter how we add rewards. We will make recent rewards less useful with *reward decay* (β), a number between 0 and 1 (but not equal to 0 or 1). The more actions it took to get to that reward, the more we will multiply that reward by β before we add it, making sure that the current reward *converges*.

Formula

To describe the current reward using a reward decay, for s our current state and a an action in Act_s, we will use the following function:

$$v_s \leq v^*_{max} * (1 - \sigma(\theta)(a)) + r(s, a) + \beta * \sum_{s' \in succ(s,a)} P(s, a)(s') * v_{s'}$$

v^*_{max} is a number larger then the highest difference between two rewards from the same state. This makes sure that, to optimize v_s, the strategy will prefer higher yielding actions.

[9] Can have as codomain any totally ordered set for the "$\leq$" operation where addition is defined. Like the rational numbers, natural numbers, etc.

3.4.2 MILP Encoding

With the reward function we can create an algorithm which creates an optimal strategy based on a *reward value* which we want to increase as much as possible.

Initiation

The algorithm will expect the following input:

- $\mathbb{P}$ - The POMDP for which we want to find an ideal strategy
- r - The reward function
- β - The reward decay, with $0 < \beta < 1$

Algorithm

Using the input defined as above, we will use the following encoding:

Algorithm 2: MILP Reward Strategy Creator

Data: $\mathbb{P}$, r

Goal: maximize v_{s_i} (maximize reward)

Constraints:

$\forall a, \theta : \sigma(\theta)(a) = 0$ or $\sigma(\theta)(a) = 1$ (deterministic strategy)

$\forall \theta : \sum_{a \in Act} \sigma(\theta)(a) = 1$

$\forall s \in S, a \in Act_s :$

$v_s \leq v^*_{max} * (1 - \sigma(\theta)(a)) + r(s,a) + \beta * \sum_{s' \in succ(s,a)} P(s,a)(s') * v_{s'}$

Variables: $\sigma(\theta)(a)$, v_s

We only need the one additional constraint and variable to make sure that this algorithm is as optimized as possible. Because we are only focused with getting as much reward as possible, we do not need to optimize a probability of arriving somewhere or worry about getting stuck. Being stuck in a rewarding loop is not so bad.

3.5 Improvements

We have now found two algorithms that are able to generate optimal strategies based on two different perspectives. Though they are correct, they do not cover all the possible scenarios[10]. Because of this, we can modify our input so that we will still be able to cover such scenarios with our algorithms.

3.5.1 Non-deterministic Strategy

Our algorithms currently only creates *deterministic* strategies. To make sure that our algorithm is also capable of creating *non-deterministic* strategies[11] we will alter the available actions before we start with the algorithm.

> *For the set of actions Act, we create a new set of actions* Act^* *with for every* $a_1, a_2, ...a_n \in Act_s$ *for some state* s *there is a* $k_1 * a_1 + k_2 * a_2 + ... + k_n * a_n = a^* \in Act_s^*$, *with* $k_1, k_2, ..., k_n \in$ {*a finite subset of [0,1]*} *and with* $\sum k_i = 1$.

By replacing the set of actions Act in the POMDP with the set of actions Act^*, we will be able to still create an *non-deterministic* strategy. We will use the k values to symbolize the distribution of potential actions.

3.5.2 Finite Paths

Our algorithm currently tries to optimize a strategy based on potentially infinitely long paths. To create an optimal strategy where only a finite amount of actions are allowed, we will create a new set of states.

> *For the set of states* S, *we create a new set of states* S_n *with for every path* $\pi \in Paths$ *with no more than* n *actions, there is a state* $s_\pi \in S_n$ *so that* $Act_{s_\pi} \sim Act_{last(\pi)}$[12] *if* π *has no more than* $n-1$ *actions. Otherwise* Act_{s_π} *is an empty set.*

Using the states S_n instead of S, we will be able to go through our algorithm the same way we did before, but this time we will be forced to stop once our path is of length n. This also works for when our path begins to repeat itself[13].

With a similar technique we could also limit specific paths. For example, we may only allow to perform a specific action so many times in a row, or force to choose an action once we have walked a certain path.

[10] Such as non-deterministic strategies and finitely many actions.

[11] Strategies where we can chose between multiple actions.

[12] With every state $last(\pi) \in S$ replaced with $s_\pi \in S_n$ instead.

[13] If no such repeats are possible, and so we would always see n distinct states after n actions, this can be simplified as cutting of all unreachable states in S.

3.6 Implementation

With our algorithms in place we can demonstrate how to implement them using actual code. To do so, we will use python 3.1 and use mip 1.9.0 [1], which allow us to build a MILP using a similar structure as our algorithms.

Should we be able to find a way to implement our algorithms using code, we will have successfully translated our abstract POMDP problem into a problem that a computer can solve, meaning that we can solve any such problems using computers from here on out.

"Knapsack" Example

To illustrate how to work with mip, they provided a small example. Here they used the model "knapsack", a model optimized to calculate the most valuable items that they can buy for a cost no larger than c:

MILP Knapsack Example

```
from mip import Model, xsum, maximize, BINARY

p = [10, 13, 18, 31, 7, 15]
w = [11, 15, 20, 35, 10, 33]
c, I = 47, range(len(w))

m = Model("knapsack")

x = [m.add_var(var_type=BINARY) for i in I]

m.objective = maximize(xsum(p[i] * x[i] for i in I))

m += xsum(w[i] * x[i] for i in I) <= c

m.optimize()

selected = [i for i in I if x[i].x >= 0.99]
print("selected items: {}".format(selected))
```

Our Implementation

Using this "knapsack" example and the algorithms described earlier, we are able to model our algorithms in the same way. We will first need to consider how we want to model our data before we can look at the specifically worked out examples.

3.6.1 Modeling A POMDP

Before we can begin with the algorithms, we first need to model the POMDP accordingly. As discussed in the "Creating An Algorithm" chapter, it is important that all our data is *finite* or we won't be able to model it. To model the POMDP, we will use a very minimalistic approach:

Objects

- **S -** A range of numbers from 0 to the amount of sites - 1.
- **Act -** A range of numbers from 0 to the amount of actions - 1.
- **Z -** A range of numbers from 0 to the amount of observations - 1.

Using just numbers we can represent any state s_n as just the number n, etc.[14]. This will make the program smoother and ultimately less complex.

Functions

- ***P*** **-** A 3d matrix P with size $S \times Act \times S$, with each value equal to $P(s,a)(s')$ or -1 if $a \notin Act_s$.
- ***O*** **-** An array O with size S, with each value equal to the corresponding observation (using numbers).

We will also create the functions $Act(s)$ and $succ(s,a)$ using P. $Act(s)$ will return all the actions that are possible from a certain state s, while $succ(s,a)$ will return all states reachable from state s using action a.

Paths

During the algorithm we won't actually be creating new paths. Yet our strategy needs to know all possible paths that it can walk. Using *finite-memory strategies*[15] we luckily only need to know a *finite amount of paths.* Before the algorithm starts we will first make these paths:

> *For a memory size m, starting with* $s_i \in Paths_m$*, for each* $\pi \in Paths_m$*, for every* $a \in Act_{last(\pi)}$ *and for every* $s' \in succ(last(\pi), a)$*, we will add* $last_m(\pi \xrightarrow{a} s')$ *to* $Paths_m$ *unless it was already added.*

We will build the $Paths$ array using the technique described and with three for-loops, with each path an object with two arrays, one for each site and one for each in-between action. After that we will derive $Paths_z$ from $Paths$, existing of the observation paths.

[14] s_i and z_i will both be equal to 0.
[15] And a finite amount of sites and actions.

Finite-memory Strategy

The main goal of both the algorithms is to find the optimal strategy. Because we are working with deterministic strategies, we can model our strategy as follows:

- ***o*** **-** A function made using two arrays, *o_index* and *o_actions*, with both size *Paths_z*. The the first array refers to the recommended action in the second array (using numbers).

Improvements

If we want to use any of the improvements mentioned earlier, like allowing for a non-deterministic strategy, or limiting all paths to a certain length, we will first need to change our set of states or actions appropriately, as described.

Implementation

Using the techniques described we can model our POMDP, paths and strategy as follows[16]:

Objects

```
S = range(...), Act = range(...), Z = range(...)
```

As described, the objects are each a list of numbers ranging from 0 to the amount of that object minus one. Because of the way ranges work in python, we don't need to subtract one from the number to create the proper range. For example, $range(4) = [0, 1, 2, 3]$.

[16] "..." is used to signify external input.

Functions

```
P = [[[... for s_ in S] for a in Act] for s in S]
O = [... for s in S]

# ================== Act(s)

def Act(s) :
    Act_s = []
    for a in Act :
        if P[s,a,0] != -1 :
            Act_s.append(a)
    return Act_s

# Act function for observations
def Act_(z) :
    Act_z = []
    for a in Act :
        for s in S :
            if O[s] == z and P[s,a,0] != -1 :
                Act_s.append(a)
                break
    return Act_s

# ================== succ(s,a)

def succ(s, a) :
    succ = []
    for s_ in S :
        if P[s,a,s_] > 0 :
            succ.append(s_)
    return succ

# succ function for any action
def succ_(s) :
    succ = []
    for s_ in S :
        for a in Act :
            if P[s, a, s_] > 0 :
                succ.append(s_)
                break
    return succ
```

The matrix and array both need to filled in before the program starts. The O array needs to be filled with numbers between zero and the amount of observations minus one (matching the list of observations). The P function needs to be filled in with probabilities (between 0 and 1) such that for every state s and action a, the sum of all the values for every site $s\_$ is one. Unless $a \notin Act_s$, in which case all values need to be minus one[17].

[17]This is to make sure that we won't make impossible actions.

Paths part 1

```
m = ...

class path:
    index = -1
    states = []
    actions = []
    obser = -1

    def last :
        return states[-1]

start = path()
start.states.append(0)
Paths = [start]

paths = Paths
new_paths = []

while len(paths) != 0 :
    for pi in paths :
        for a in Act(p.last) :
            for s in succ(pi.last,a) :
                pi_new = path()
                pi_new.states = pi.states + s
                pi_new.actions = pi.actions + a

                pi_new.states = pi_new.states[-m:]
                pi_new.actions = pi_new.actions[-m+1:]

                if pi_new not in Paths :
                    new_paths.append(pi_new)

    Paths.extend(new_paths)
    paths = new_paths
    new_paths = []

for i in range(len(Paths)) :
    Paths[i].index = i

...
```

Paths part 2

```
...

# Paths were we only know of the observation

Paths_z = []

for pi in Paths :
    theta = path()
    theta.actions = pi.actions

    for s in pi.states :
        theta.states.append(O[s])

    found_theta = False
    for i in range(len(Paths_z)) :
        if Paths_z[i] == theta :
            pi.obser = i
            found_theta = True

    if !found_theta :
        theta.index = len(Paths_z)
        pi.obser = theta.index
        Paths_z.append(theta)
```

Using a predefined m value we can actually make our algorithm walk every possible path before we start. Every time we add something new to one of our recently created paths, we will shrink it down to the size of m, with m states and $m-1$ actions. As soon as no new paths follow from the recently created paths will we stop.

After creating every path in *Paths*, we can then convert our array of paths to an array of *observation paths*, the one our strategy uses. We will build a connection between any path and an observation path using the *obser* variable. For any path *pi* in *Paths*, its observation path is equal to $Paths\_z[pi.obser]$.

Using the *Paths_z* and *Act* arrays as defined earlier, we can create an implementation of our strategy:

Strategy

```
o_index = [-1 for theta in Paths_z]

def o(theta_index) :
    index = o_index[theta_index]
    if index == -1 :
        return -1
    else :
        actions = Act_(Paths[theta_index].last)
        return actions[index]
```

For any possible path in the observation paths, we can define a strategy or a recommendation for the proper action that we should take. We will use the *o* function to give the proper return value, where if there are no actions possible, *o* will return a non-existing action (-1).

We use a method where we build *o* like a small library, whereby the value of the array o_index refers to the index of the action that we should take. Though not pretty, this allows us to work with the different possible actions for each observation[18].

Combined Implementation

If we bring these implementations together in one large class or file we will have successfully modelled our POMDP in code. We could then refer to these pieces of code using such a class for easier use. For now, to keep the following algorithms brief, we will simply refer to these implementations by using comments.

[18] Alternatively, we could memorize the array of all the possible actions from any path, so that this process may go quicker.

3.6.2 Success States Implementation

Using the input as defined earlier, we can now implement our success states algorithm. Additionally, because we working with success and failure states, we will need to implement both S_S and S_F. We will use arrays for both of them, each with numbers of their corresponding states, according to S.

MILP Success Strategy Creator part 1

```
from mip import Model, xsum, maximize, INTEGER,
    CONTINUOUS

# ========== Initiation

# ... set objects ...
# ... set functions ...
# ... set paths ...
# ... set strategy ...

SS = [...] # Set according to our scenario
SF = [...] # Set according to our scenario

S_star = []

for s in S :
    if (s not in SS) and (s not in SF) :
        S_star.append(s)

# ========== Variables & Goal

z = m.add_var(name='zCost', var_type=INTEGER, lb=-10,
    ub=10)

m = Model("SuccesStrategyCreator")

p = [m.add_var(var_type=CONTINUOUS, lb=0, ub=1) for s
    in S]
r = [m.add_var(var_type=CONTINUOUS, lb=0, ub=1) for s
    in S]
t = [[m.add_var(var_type=INTEGER, lb=0, ub=1) for s_ in
     S] for s in S]

for theta in Paths_z :
    possible_actions = len(Act_(theta.last))
    if possible_actions > 0 :
        o_index[theta.index] = m.add_var(var_type=
            INTEGER, lb=0, ub=possible_actions-1)

m.objective = maximize(p[0]) # Goal

...
```

MILP Success Strategy Creator part 2

```
...

# ================== Constraints

# p[s] constraints
for s in SS : m += p[s] == 1
for s in SF : m += p[s] == 0
for pi in Paths :
    if pi.last in S_star :
        s = pi.last
        for a in Act(s) :
            m += p[s] <= (1 - (o(pi.obser)==a)) +
                sumx(P[s,a,s] * p[s] for s in succ(s,a))

# r[s] constraints
for s in S_star :
    if len(Act(s)) == 0 :
        m += r[s] == 0
for s in S_star :
    for s_ in succ_(s):
        m += r[s] < (1 - t[s,s_]) + r[s_]
for pi in Paths :
    if pi.last in S_star :
        s = pi.last
        for a in Act(s) :
            m += p[s] <= (1 - (o(pi.obser)==a)) +
                sumx(t[s,s_] for s_ in succ(s,a))

# ==================

m.optimize()
```

As can be seen straight away, the structure of the implementation largely follows that of the previous algorithms. Instead of "$\forall x \in X$" we use "$for\ x\ in\ X$:" here[19]. Similarly, any otherwise normal assumption needs to be written out in full if it needs to work on a computer.

Using The *mip* Model

To work with the mip library we need to add all the customizable variables to the model using the *add_var* function. Similarly we need to add all the constraints to the model by using $+=$. Once we have all our variables and constraints set we can use $m.optimize()$ to find the optimal solution.

[19] Additionally, in the program we make handy use of the fact that in python $True$ is equal to 1 and $False$ is equal to 0. Because of this, we can simply use "$o[theta] == a$" to get the values zero or one that we need.

3.6.3 Reward Implementation

In the same way as before we can now implement our algorithm using the input as defined earlier, only now we want to maximize our reward instead.

MILP Reward Strategy Creator

```
from mip import Model, xsum, maximize, INTEGER,
    CONTINUOUS

# ========== Initiation

# ... set objects ...
# ... set functions ...
# ... set paths ...
# ... set strategy ...

B = ... # Set B according to our scenario
r = [[... for a in Act(s)] for s in S] # Set r
    according to our scenario
v_max = ... # Set v_max according to the expected max

# ========== Variables & Goal

m = Model("RewardStrategyCreator")

v = [m.add_var(var_type=CONTINUOUS, lb=0, ub=1) for s
    in S]

for theta in Paths_z :
    possible_actions = len(Act_(theta.last))
    if possible_actions > 0 :
        o_index[theta.index] = m.add_var(var_type=
            INTEGER, lb=0, ub=possible_actions-1)

m.objective = maximize(v[0]) // Goal

# ========== Constraints

for pi in Paths :
    if pi.last in S_star :
        s = pi.last
        for a in Act(s) :
            m += v[s] <= v_max*(1 - (o(pi.obser)==a)) +
                r[s,a] + B * sumx(P[s,a,s_] * v[s_]
                for s_ in succ(s,a))

# ==========

m.optimize()
```

Similar to our previous implementation, we start with our initial input, then set all our variables and constraints using the methods defined by the mip library, and then calculate the optimal solution accordingly. Because this implementation only uses one constraint, the code is a lot smaller than the previous one.

3.6.4 Practical Use

Now that we got both of our algorithms implemented using python and the mip library [1], we can start to usw these algorithms to calculate various optimal solutions to many different POMDP problems, as discussed in the introduction.

This practical example of how we can find a solution to any POMDP problem using a now actually working MILP implementation demonstrates how well these two concepts are connected, and how problems involving POMDPs are easily reducible to such a mathematical way of programming we well understand.

Visual Implementation

To make this all really work nicely, we have implemented a program which will help visualize the algorithm. We will be able to manually set the variables accordingly and so create a program which can easily be used. This allows for an even greater conformation that any POMDP problem can be solved by *anyone*.

Chapter 4

Related Work

In this thesis we largely rely on just a few papers. Since this is mostly theoretical research which can be done using just the definitions, we did not need many sources. However, this paper is does largely draw inspiration from other similar efforts to find optimal solutions to POMDP problems.

4.1 Underlying Theory

For the underlying theory used in this thesis we rely on just a few papers. These papers provide the basic definitions that we were using.

Strengthening Deterministic Policies for POMDPs

This thesis was not possible if not for the large scale support from the paper *Strengthening Deterministic Policies for POMDPs* [17]. From this paper a large amount of the definitions are derived, and even the algorithms are largely based on that paper.

Permissive Finite-state Controllers of POMDPs via Parameter Synthesis

Permissive Finite-state Controllers of POMDPs via Parameter Synthesis [12] shows numerous definitions that we also use here, among them the term *strategy* and the definition for a *path*. The theory behind *distributions* is largely based on that paper, though slightly simplified.

Human In The Loop Synthesis For Partially Observable Markov Decision Processes

The paper *Human In The Loop Synthesis For Partially Observable Markov Decision Processes* [5] is probably the most explanatory paper on this topic.

4.2 Other Implementations

Apart from our own implementation, there are also a number of other methods to try to automatically find a solution to any POMDP problem.

Using MILPs

In the paper *Mixed Integer Linear Programming For Exact Finite-Horizon Planning In Decentralized Pomdps* [2] they provide an alternative method to try and relate a specific kind of POMDP to MILPs. They then are able to formulate a MILP algorithm capable of solving such POMDP problems. However, their underlying theory is quite different than that of ours, which means that their algorithms may not be useful for us, and visa versa.

Using (Recurrent) Neural Networks

Various papers such as *Counterexample-Guided Strategy Improvement for POMDPs Using Recurrent Neural Networks* [4] and *Verifiable RNN-Based Policies for POMDPs Under Temporal Logic Constraints* [3] suggest the use of a neural network to train a program to find the optimal strategy. This method would then highly rely on the same techniques used in AI today.

Using Deep Learning

Slightly different from using neural networks, papers such as *Deep Variational Reinforcement Learning for POMDPs* [10] and *QMDP-Net: Deep Learning for Planning under Partial Observability* [13] suggest using deep learning instead.

Other Code Implementations

Other then our own implementation, there have been other efforts to try to create actual programs which allow for POMDP problems to be automatically solved. Among them is the *Package 'pomdp'* [14]. This implementation (available on GitHub) provides an alternative to our method by using a *completely different* method.

Similar papers

Other implementations include *Parameter Synthesis for Markov Models* [11], which mostly deals with just MDPs, *Strategy Synthesis in POMDPs via Game-Based Abstractions* [16], which solves the problem by looking at it using a game perspective, and finally *Value-Function Approximations for Partially Observable Markov Decision Processes* [9], which mostly examines the efficiency of various algorithms.

Chapter 5

Conclusions

5.1 Conclusion

In our research we saw that there is a definitive connection between the calculation of optimal strategies and the computation of MILPs. We were able to link the two together using MILP algorithms, and were then, as proof of concept, able to show that these algorithms can be implemented in the form of actual working code using python and the mip library [1].

Though our research has only covered a small section of the many possibilities of POMDP scenarios, we can be confident that this encoding can serve as a step towards solving any possible POMDP scenario in a very efficient manner. Should we reach that point, we will be able to solve any problem automatically as long as it can be described as a POMDP.

5.2 Further Research

Many more areas or research are still not fully researched, Among them for example the possibility of prioritizing speed over success rate, or desiring a higher reward per action on average instead of a high reward in total.

5.3 Acknowledgement

This paper was not possible if it wasn't for the incredible support from Nils Jansen to keep on going. Though this has been an enormous project, it has definitely been guided to finally a good result thanks to the spirit of not yielding till it is done.

Bibliography

[1] Python-MIP tool. http://python-mip.com/.

[2] Raghav Aras, Alain Dutech, and François Charpillet. *Mixed Integer Linear Programming For Exact Finite-Horizon Planning In Decentralized Pomdps*. 2007. https://arxiv.org/abs/0707.2506.

[3] Steven Carr, Nils Jansen, and Ufuk Topcu. *Verifiable RNN-Based Policies for POMDPs Under Temporal Logic Constraints*. 2020. https://arxiv.org/abs/2002.05615.

[4] Steven Carr, Nils Jansen, Ralf Wimmer, Alexandru C. Serban, Bernd Becker, and Ufuk Topcu. *Counterexample-Guided Strategy Improvement for POMDPs Using Recurrent Neural Networks*. 2019. https://arxiv.org/abs/1903.08428.

[5] Steven Carr, Nils Jansen, Ralf Wimmer, Jie Fu, and Ufuk Topcu. *Human In The Loop Synthesis For Partially Observable Markov Decision Processes*. 2018. https://arxiv.org/abs/1802.09810.

[6] Iadine Chadès, Eve McDonald-Madden, Michael A. McCarthy, Brendan Wintle, Matthew Linkie, and Hugh P. Possingham. *When to stop managing or surveying cryptictthreatened species*. 2008. https://www.pnas.org/content/105/37/13936.

[7] Marek Grzes, Jesse Hoey, Shehroz S. Khan, Alex Mihailidis, Stephen Czarnuch, Dan Jackson, and Andrew Monk. *Relational approach to knowledge engineering for POMDP-based assistance systems as a translation of a psychological model*. 2020. https://www.sciencedirect.com/science/article/pii/S0888613X13000662.

[8] Trung H. Bui, Mannes Poel, Anton Nijholt, and Job Zwiers. *A POMDP approach to Affective Dialogue Modeling*. https://wwwhome.ewi.utwente.nl/~anijholt/artikelen/ios2007-2.pdf.

[9] Milos Hauskrecht. *Value-Function Approximations for Partially Observable Markov Decision Processes*. 2000. https://jair.org/index.php/jair/article/view/10262.

[10] Maximilian Igl, Luisa Zintgraf, Tuan Anh Le, Frank Wood, and Shimon Whiteson. *Deep Variational Reinforcement Learning for POMDPs*. 2018. https://arxiv.org/abs/1806.02426.

[11] Sebastian Junges, Erika Abraham, Christian Hensel, Nils Jansen, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. *Parameter Synthesis for Markov Models*. 2019. https://arxiv.org/abs/1903.07993.

[12] Sebastian Junges, Nils Jansen, Ralf Wimmer, Tim Quatmann, Leonore Winterer, Joost-Pieter Katoen, and Bernd Becker. *Permissive Finite-state Controllers Of POMDPs Via Parameter Synthesis*. 2018. https://arxiv.org/abs/1710.10294.

[13] Peter Karkus, David Hsu, and Wee Sun Lee. *QMDP-Net: Deep Learning for Planning under Partial Observability*. 2017. https://arxiv.org/abs/1703.06692.

[14] Package 'pomdp' team. *Package 'pomdp'*, 2020. https://cran.r-project.org/web/packages/pomdp/pomdp.pdf.

[15] Matthijs T.J. Spaan. *Cooperative Active Perception using POMDPs*. 2008. https://www.semanticscholar.org/paper/Cooperative-Active-Perception-using-POMDPs-Spaan/535ca58f58838ce085493f167f81d40ed306b61b.

[16] Leonore Winterer, Sebastian Junges, Ralf Wimmer, Nils Jansen, Ufuk Topcu, Joost-Pieter Katoen, and Bernd Becker. *Strategy Synthesis in POMDPs via Game-Based Abstractions*. 2019. https://arxiv.org/abs/1708.04236.

[17] Leonore Winterer, Ralf Wimmer, Nils Jansen, and Bernd Becker. *Strengthening Deterministic Policies for POMDPs*. 2020. https://arxiv.org/abs/2007.08351.

Appendix A

Appendix

A.1 MDP Coin-toss Example

We can describe a coin-tossing scenario as an MDP. Our states will be *heads* and *tails* and we will begin with *heads*. We have only have one action, *"toss"*[1], and the probabilistic function will say that, no matter in what state we are, performing the "toss" action will give a 50% chance of ending up with heads and a 50% chance of ending up with tails.

Figure A.1: MDP Coin Example

$$\mathbb{M}_{coin} = (S_{coin}, s_{i\,coin}, Act_{coin}, P_{coin})$$

with:
$S_{coin} = \{heads, tails\}$,
$s_{i\,coin} = heads$,
$Act_{coin} = \{toss\}$,
P_{coin} is defined such that:
$P_{coin}(heads, toss) = P_{coin}(tails, toss) = (0.5 : heads, 0.5 : tails)$

In our MDP $\mathbb{M}_{coin}$ (A.1) we can see the coin sides *heads* and *tails* as our states (S_{coin}), *heads* as the begin state ($s_{i\,coin}$), the *toss* action as the only action (Act_{coin}) and the probabilistic function which simulates a coin toss (P_{coin}).

Starting from the state *heads*, we can perform the action *toss* to randomly end up in a new state, either *heads* or *tails*, each with a 50% probability. From that state we could again do the *toss* action to end up in yet a new state. Note that it is possible to end up in the same state after performing the toss action.

[1] A MDP with only one action can also be described as a Markov Chain

A.2 POMDP Coin-toss Example

As an example of a POMDP we could extend our coin-tossing scenario with the condition that we *can not look at the coin after we threw it*[2]. We can use our MDP like before[3], with the addition of the observation *"mystery-side"*, and a function mapping both heads and tails to this hidden *"mystery-side"*.

Figure A.2: POMDP Coin Example

$$\mathbb{P}_{coin} = (M_{coin}, Z_{coin}, O_{coin})$$

with:
$M_{coin} = \mathbb{M}_{coin}$
$Z_{coin} = \{mystery\text{-}side\}$,
O_{coin} is defined such that:
$O_{coin}(heads) = O_{coin}(tails) = mystery\text{-}side$

With this POMDP $\mathbb{P}_{coin}$ (A.2) we can see our previous MDP with everything that we got from there (M_{coin}), the single observation *"mystery-side"* (Z_{coin}) and the function which maps both *heads* and *tails* to this *"mystery-side"* observation (O_{coin}).

Similar as before, we can perform actions from our begin observation, *"mystery-side"*. From there on, tossing the coin will only give the same *"mystery-side"* observation back, while which side of either *heads* or *tails* truly came up we do not know.

[2]Maybe we are still covering the coin with our hand
[3]See figure A.1 MDP Coin Example