Planning for Money Laundering Investigations

Master's Thesis

Max Weltevrede
Student number: 4754166

Algorithmics Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
to be defended publicly on

Supervisors:
Dr. M.T.J Spaan (TU Delft)
Dr. M. Alos Palop (ING)

October 13, 2020

Preface

This thesis was written as part of my Master's Thesis Project conducted at the Algorithmics group of the Delft University of Technology. About a year ago I was asked by my supervisor Dr. Matthijs Spaan if I was interested in a graduation project in collaboration with ING. The idea behind the collaboration was to investigate the possibility of optimizing the anti-money laundering process within a financial institution such as ING. This was motivated by a study on the maximization of influence spread in a social network of homeless youth living in big cities.

We conceptualized a similarity between on the one hand the selection of homeless youth intervention participants based on their social network and on the other hand the selection of investigation subjects inside a financial institution based on a transaction network. As far as we knew this similarity was not conceptualized before. Therefore, the project consisted of a study and modeling of the problem domain as well as investigating and designing solution methods.

I would like to thank several people for their advice and support in completing this work. First of all, I would like to thank my supervisors and members of my thesis committee Dr. Matthijs Spaan (TU Delft) and Dr. Mireia Alos Palop (ING) for their guidance throughout this project. I would also like to thank the other member of my thesis committee Dr. Frans Oliehoek, for providing feedback and evaluating my work. In the latter half of the project, I continually received very helpful guidance from Canmanie Ponnambalam of the Algorithmics group, for which I am very grateful. At ING I would also like to thank Fabian, Rafah and Luiz for their helpful feedback and their listening ear. Finally, I am very grateful for my family and friends for their continuous support in all my endeavours, including this project.

Max Weltevrede
Delft, the Netherlands
October 13, 2020

Abstract

According to the United Nations, the amount of money laundered worldwide each year is an estimated 2 - 5% of global GDP (equivalent to $800 billion to $2 trillion in US dollars). This is money that criminal enterprises rely on to operate. For that reason, the European Union demands that gatekeepers (banks and other obliged entities) apply measures to counteract money laundering. Current industry state of the art anti-money laundering (AML) techniques ultimately revolve around investigations by specialized financial investigators of suspicious behaviour. Due to the human nature of this work, this process is relatively slow and has limited capacity. Deciding in the most optimal way what financial entities to investigate and when is not a trivial problem. However, optimizing this sequential decision making problem could significantly decrease the time-scale in which fraudulent actors are caught. This thesis will formulate the AML problem as a Partially Observable Markov Decision Problem. It will design and implement an AML model and investigate the challenges associated with optimizing it. In particular, several Partially Observable Monte-Carlo Planning based methods are proposed that exploit the combinatorial structure of the actions to overcome the challenges associated with a large action space. The methods are empirically evaluated on the AML problem and compared to a baseline solution. The results indicate that exploiting the combinatorial structure increases the performance in this problem scenario. However, it seems that exploiting the structure to the highest degree does not always lead to the best performance. Additionally, we show that the proposed methods can match or even outperform the upper bound set by the baseline solution.

Contents

Part I

Introduction

Chapter 1

Introduction

According to the United Nations, the amount of money laundered worldwide each year is an estimated 2 - 5% of global GDP (equivalent to $800 billion to $2 trillion in US dollars)[1]. This is money that criminal enterprises rely on to operate. For that reason, the European Union demands that gatekeepers (banks and other obliged entities) apply measures to counteract money laundering. In recent history, failure to oblige to this standard has led to large fines for certain financial institutions. In 2018 ING, a Dutch multinational banking and financial services corporation, settled with the Netherlands Public Prosecution Service (Openbaar Ministerie) on a sum of 775 million euro for negligence in combating money laundering. Money laundering is a global problem but the practice of combating it, also called Anti-Money Laundering (AML), is far from trivial. In order to properly perform AML practices, one has to detect fraudulent transaction behaviour of individual accounts based on (most likely) partial knowledge of a network of immense scale. The amount of data is so large that it would be impossible to manually check every single suspicious transaction. The problem amounts to finding a needle (fraudulent behaviour) in a haystack (transaction graph) with only very limited resources.

1.1 Anti-Money Laundering

In the context of a financial institution, AML translates to detecting and reporting fraudulent behaviour. This is however easier said than done. In a financial institution such as a bank, detecting fraudulent behaviour requires monitoring immense transactional data flows for 'anomalous' activity. For a moderately sized bank in a country like the Netherlands, this can range up to millions of accounts and even billions of transactions. However, this would only give a partial view of the entire transactional network of the country. Money laundering typically doesn't restrict itself to a single financial institution. So, in order to properly combat money laundering, transactional views of multiple different financial institutions would have to be combined, increasing the scale of the problem even further.

1.1.1 Rule-based Systems

At this point, it will be clear that AML cannot be a solely manual process. Up to very recently, financial institutions would typically try to detect fraudulent behaviour using a primitive rule-based system. There would be certain human-crafted or curated rules that once broken would trigger a manual investigation. An example of a rule could be a single transaction that exceeded a certain large amount of money. One of the issues with this system is that if a criminal knows the rules, it is very easy to avoid breaking them. Another is that these rules typically are a very crude way of detecting fraudulent behaviour. In order to avoid illicit activity going undetected, the thresholds for triggering an investigation have to be set low. This results in a large amount of false positive triggers which exceeds the (manual) investigation capacity.

1.1.2 Machine Learning

At the moment, many financial institutions are looking towards machine learning for a solution. One example would be to use some form of supervised learning or anomaly detection to derive a so called *risk score* for every entity in the financial institution in question. This risk score would denote the likelihood of an entity engaging in illicit behaviour: the higher the risk score, the higher the probability of the entity being fraudulent. A small group of specialized investigators could then perform in-depth investigations into the entities in descending order of risk score. The key idea is that machine learning can hopefully provide a more sophisticated method of detecting illicit behaviour compared to the primitive, heuristic rule-based system.

Another way of viewing this approach is as a classifier that learns to classify illicit behaviour. The confidence in the classification of an entity is then directly proportional to the risk score of said entity.

1.1.3 Sequential Decision Making

The goal of the classifier approach described above is to detect suspicious behaviour of an entity. However, detecting suspicious behaviour is only the first step of the AML process. If a financial institution suspects an entity of laundering money, an investigation into this entity has to be launched. The goal of this investigation is to confirm the suspicion and simultaneously collect evidence against the illicit behaviour of this entity. Only if the suspicion is confirmed beyond reasonable doubt and enough evidence has been collected, can the entity be reported to the Public Prosecution Service. This is the final step of the AML process within a financial institution.

This second part of the AML process (that involves the investigations) can be

modelled as a *sequential decision making* problem. A sequential decision making problem is a problem in which decisions (usually performing an action) have to be made in a successive order (one after another). The 'sequential' in sequential decision making means that this framework explicitly considers the dynamics of the problem and its dependency on the decisions being made throughout time. Examples of sequential decision making applications are driving a car, managing resources or playing a game.

1.2 Current AML Research

Current AML research is mostly focused on applying machine learning techniques to detect fraudulent behaviour and structures in large datasets. In a review of machine learning techniques for AML [2], the authors provide a classification of algorithms that identifies several different approaches. This classification includes approaches such as *AML typologies* (detecting behaviour similar to known money laundering cases), *link analysis* (detecting links between entities), *risk scoring* (rank entities by potential risk) and *anomaly detection* (identify deviation from the normal transactional behaviour).

All of the identified approaches in [2] focus on classifying illicit behaviour rather than optimizing the investigation process. One example of this is [3], in which the authors show the importance of exploiting the graph structure of the transactional network. They show that enhancing the local node features with features aggregated from a neighbourhood around the node increases classification performance in all of their tested methods. Additionally, they show that adding node embeddings computed by a Graph Convolutional Network (GCN) increases performance further. This strongly indicates the importance of the graph structure in the classification of illicit behaviour. It is worth noting that this paper also introduces the Elliptic Data Set, which is (to their knowledge) the largest labelled transaction data set publicly available in any cryptocurrency. This data set includes time slices of Bitcoin transactions in the order of a couple of thousand nodes and edges (over 200k nodes and edges for the entire dataset). This still is a relatively small network compared to the real transaction network of a financial institution such as ING.

Most AML research is performed on synthetic data sets, such as the ones generated from the AMLSim simulator [4]. In this paper, the authors compare the scalability of different GCN methods on a large synthetic transaction graph. They show that current GCN methods can struggle to scale up to large, real-sized networks. In [5], they also propose a GCN method that handles the dynamics of a changing transactional graph (as new transactions are being made every day). These papers illustrate that a lot of the focus in AML research is on improving the performance and scalability of the classification of fraudulent behaviour.

Another interesting area of research is looking beyond just the transactional data. In [6], the authors identified three major risk factors in order to create three separate graphs: a Geographical Area Network, an Economic Sector Network and a Transaction Network. They then apply social network analysis to these networks and show some of these are very good predictors of a high-risk score. This study shows that data other than the transactional network can be very relevant for AML.

1.3 Problem Description

To our knowledge, none of the AML research considers the sequential decision making aspect of the investigation process. However, this is a substantial part of the AML process that could play a significant role in the overall objective of combating money laundering. We will provide here an informal definition of the AML process as a sequential decision making problem.

1.3.1 Informal Definition

The AML process can be framed as a sequential decision making process by considering the investigations as the decisions that have to be made. We will assume that the structure of this process is roughly equivalent to the following:

1. **Detect Suspicious Behaviour** - Derive a risk score for every entity in your network.
2. **Investigate Suspicious Behaviour** - Choose a set of entities to investigate.
3. **Repeat With New Information** - Repeat the process with new information obtained from the investigations.

If we assume every investigator can investigate a single entity, the size of the set of entities in step 2 is equal to the number of investigators. Due to the size of the network, the number of investigators will typically be small compared to the number of potentially suspicious entities in that network. Therefore, multiple sequential repetitions of this process will be required.

If an investigation reveals the suspicious behaviour of an entity, this will result in a case file containing evidence of this behaviour. If the case file is substantial enough, it will be sent to the Public Prosecution Service and result in a so-called Suspicious Activity Report (SAR). The goal of the decision making process is to get a maximal amount of SARs in a minimal amount of time possible.

1.3.2 Motivation

It is not necessarily trivial to find the most optimal sequence of investigations that maximizes SARs in a minimal amount of time. In the presence of a risk score provided by some classifier, an intuitive approach would be to sequentially investigate the suspicious entities in descending order of the risk score. However, this sequence might not always lead to an optimal order of investigations. We will argue this with an example.

Figure 1.1 shows a scenario that might occur at a financial institution. In it, we see a toy example of a transaction network where nodes are accounts and edges financial transactions. In this example are three nodes (accounts) with a particularly high-risk score. Node 1 having the highest risk, followed by 2 and 3 (in that order). This scenario involves two investigators that can perform investigations into nodes in parallel.

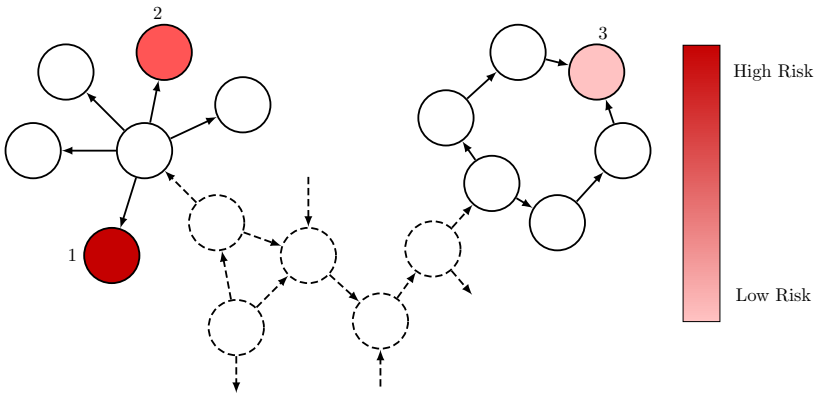


Figure 1.1: An example scenario indicating the importance of the investigation dynamics. The investigations into node 2 and 1 are most likely related to each other due to their proximity in the network.

The intuitive approach

The intuitive approach mentioned above would investigate nodes in order of descending risk. This approach would therefore investigate nodes 1 & 2 first (and in parallel). However, an in-depth investigation into an entity does not involve only the entity under investigation. It typically also involves an analysis and investigation into entities that it has business with (close proximity in the transaction network). So, an investigation into node 1 most likely involves an analysis/partial investigation into node 2 (and vice-versa).

Why we need sequential decision making

What the optimal order of investigation is, depends on the dynamics of the investigation process. For instance, it might be that node 1 and 2 have a very high risk due to some error in the classification process (they are not actually fraudulent). Since they are in the same part of the transaction network, they might have erroneous scores due to a shared underlying cause. Instead of investigating nodes 1 & 2, it might be more efficient to investigate nodes 1 & 3 (or 2 & 3). Investigating node 1 might unveil the underlying cause of its erroneous classification. Once this underlying cause is fixed it can also fix the erroneous classification of node 2, removing the need for an investigation in node 2 altogether.

On the other hand, it might be the case that investigating nodes 1 & 2 in parallel is actually more efficient. Perhaps the investigators can benefit from cooperation in investigating the shared neighbourhood of nodes 1 & 2. Or inversely, investigating nodes 1 & 2 at the same time might cause duplicated work in some way between the two investigators. Either way, it illustrates that the characteristics and dynamics of the investigation process can play an important role in optimising this part of the AML problem.

Moreover, the investigations in the AML process are performed by human investigators specialized in financial crimes. They can take anywhere from days up to multiple months to complete. Therefore, a small shift in the sequence of investigations could potentially result in days or even months of delay. This is why this thesis will focus on the sequential decision making aspect of the AML process and will investigate the possibility of optimizing it.

1.3.3 Problem Frameworks

A similar problem domain to ours is that of *active learning*. Active learning is a machine learning framework in which a learning algorithm is able to interactively query some information source to obtain labels at new data points [7]. The active learning framework and our problem share the principle research question: *"Which instances to query and when?"*. However, active learning algorithms typically try to find those instances to query that will result in the largest improvement in classification. Our objective is more akin to querying as many instances of a particular class (the illicit class) as possible, regardless of whether this query will improve classification or not.

Additionally, in active learning it is assumed that the queried labels are always 100% accurate. In our case, an AML investigation does not necessarily return the correct label with certainty. In fact, the results of an investigation depend on what instances were previously investigated. We require a framework that takes into account that the outcome of an action might depend on previous actions taken. This is what we refer to as the dynamics of the investigation process.

A framework capable of modelling the dynamics of a system is the Markov Decision Process (MDP) framework. The MDP framework assumes the user knows the state of the system at all times. In the AML process, the state of the system would include whether an account is illicit or not (since the result of an investigation depends on this). However, it is exactly the objective of the AML process to attain this knowledge and is therefore not "known to the user at all times".

Therefore, the framework used in this thesis is the Partially Observable Markov Decision Process (POMDP) framework. This is a generalization of the MDP framework to problem domains for which users do not have to know the underlying state of the system. Instead, the POMDP framework models problems in which the user can only attain (partial) information on the state of the system (whether an account is illicit) through *observations* (investigations) of said system.

We refer to appendix B for a continued discussion on similar problem domains/frameworks.

1.4 Challenges

In modelling the AML process in the POMDP framework, there are a number of challenges one can encounter. Some of those challenges are related to the size of the problem. Compared to typical POMDP problems, the AML problem has a relatively large scale. The scale of a POMDP problem is typically measured in three ways: the number of possible states the system can have, the number of possible decisions the user can make and the number of possible observations the user can receive. All of those provide their set of challenges. There is less focus in the POMDP research on the issue of a large amount of possible decisions the user can make, which will be a focus of this thesis.

Large amount of possible decisions

In a sequential decision making problem, at every time step we want to find what decision is optimal to make at that moment. However, if there are a lot of decisions to choose from, it might not be possible to consider all of them, increasing the chances of missing the optimal decision. This problem only becomes worse when optimizing over an entire sequence of future decisions to make. In fact it increases exponentially with how far you want to look into the future [8]. This is why a large amount of possible decisions to make can pose challenges in a sequential decision making process and in particular in POMDPs.

In our case, a decision is a subset of the nodes in the transactional graph. The size of this subset is equivalent to the number of specialized investigators there

are available. The amount of possible decisions one can make is large due to its combinatorial nature: the number of possible subsets of size K of an N node network is $\binom{N}{K}$. In this thesis, we will try to exploit this combinatorial nature to help us overcome the challenges associated with the large amount of possible decisions.

Modelling

A completely different challenge is related to the modelling of the investigation process. All solution approaches in the POMDP framework require some form of a model of the process that it tries to optimize. That means we require some sort of model of the investigation process in AML. Because there is no prior research that considers AML as a sequential decision making process, there does not yet exist a model we can use. So, part of this thesis is also about how we define our model of the investigation process.

1.5 Objective

The objective of this thesis will be to frame the anti-money laundering process as a sequential decision making process and optimize it. The starting point of this thesis will be a classifier that can produce a risk score for all our entities. It will assume the existence and availability of such a classifier and use it to develop and evaluate sequential decision making algorithms.

In general, the main objective of this thesis can be summarized with the following research question:

Main Research Question - Can Anti-Money Laundering practices be improved by sequential decision making algorithms?

Given the challenges mentioned in the previous section, this research question can be further specified. We identify three sub research questions:

Research Question 1 - Is it possible to model the AML decision making process in the POMDP framework?

As was mentioned in the previous section, we will have to define a model for the AML process ourselves. Defining a model in the POMDP framework requires defining concepts (such as states, actions, observations, transition probabilities and more) that have to adhere to certain restrictions. This thesis will investigate the best possible way to define these concepts for the AML problem.

The AML POMDP can have a relatively large scale in terms of possible states, possible decisions *and* possible observations. We need to investigate which POMDP solution approaches can overcome the challenges associated with these properties and/or how they can be adjusted to do so. It is likely that the existing methods are not specific to an action space structured in the way ours is. This leads to the second question:

Research Question 2 - How can we exploit the combinatorial structure of the possible decisions to improve the performance of POMDP solution approaches?

Our action space is highly structured in a predictable way. We need to investigate how and when to use this structure to guide our search for the optimal solution. The structure is due to the combinatorial nature of the actions. This combinatorial structure arises whenever a POMDP has an action that consists of selecting a subset (of a particular size) out of a larger set. A POMDP solution approach exploiting this structure is hopefully applicable to any POMDP characterized by this property of the action space. Our third research question is:

Research Question 3 - How do sequential decision making approaches compare to methods that ignore the investigation dynamics?

The ultimate goal of AML is to file a maximal amount of SARs in a minimal amount of time. We need to investigate how framing it in a sequential decision making context affects this goal compared to a method that ignores the investigation dynamics. This should give an indication whether framing the AML process in this way is worth the effort.

1.6 Main Contributions

In answering the research questions defined above we identify the following main contributions of this thesis. To our knowledge, no other research frames the AML problem as a sequential decision making process. Our first contribution therefore is to define the AML process as a Partially Observable Markov Decision Process (POMDP). For the same reason we also have to define a POMDP transition and observation model and create and implement a simulation for it. We identify the challenges associated with this POMDP and perform a literature study on POMDP solution approaches to overcome those challenges.

Additionally, we adjust the implementation of the Partially Observable Monte-

Carlo Planning (POMCP) algorithm to deal with the memory bottleneck caused by the large amount of possible decisions. This is then combined with the partitioning of the decisions into sub-decisions, inspired by literature from similar problem domains. We exploit the symmetry of the problem by adding to this the notion of transpositions, turning our search tree into a search Directed Acyclic Graph (DAG). We then apply an existing search DAG framework to the sub-decisions of our adapted POMCP algorithm and compare it with a different approach that is designed to maximally exploit the transpositional information available.

Finally, this thesis performs an empirical evaluation of the different methods proposed and compares it with a baseline approach in AML practices.

1.7 Outline

Following this introduction chapter are two background chapters: chapter 2 and chapter 3. Chapter 2 contains background information on the Partially Observable Markov Decision Process (POMDP) and its solution approaches that will be used in this thesis. Chapter 3 expands on this by considering some approaches that can take into account the structure of the action space.

Chapters 4 and onward contain the contributions of this thesis. Chapter 4 defines the formal POMDP definitions of the AML process. It is followed by chapter 5 that discusses all the solution approaches proposed in this thesis.

Chapters 6 through 9 detail the empirical evaluation of the methods proposed. The methodology can be found in chapter 6, the experimental setup in chapter 7, results in chapter 8 and discussion in chapter 9. Finally, chapter 10 contains the conclusion and discusses recommendations, limitations and areas of future research.

Part II

Background

Chapter 2

Partially Observable Markov Decision Process

This chapter contains the theoretical background on Partially Observable Markov Decision Processes (POMDP) that will be used throughout the rest of this thesis. A POMDP is a generalization of the Markov Decision Process (MDP) to partially observable environments. This chapter will start with a brief description of a Markov decision process in section 2.1. Following this will be a description of a Partially Observable MDP (POMDP) together with a discussion of possible POMDP solution approaches in section 2.2. Section 2.3 will detail the Partially Observable Monte-Carlo Planning (POMCP) algorithm, which is the particular POMDP solution approach used in this thesis.

2.1 Markov Decision Process

Formally, an MDP is a discrete-time stochastic control process. It is called a *Markov* decision process because it is an extension of a so-called Markov chain.

2.1.1 Markov Chain

A Markov chain is a stochastic model that describes a sequence of events that transition the system from one state to another. In a Markov chain, the transitions between states satisfy the *Markov property*. This property specifies that the probability of transitioning to state s_{t+1} at time $t+1$ only depends on the state s_t at time t, not any previous states. In other words

$$P(s_{t+1}|s_1, s_2, ..., s_t) = P(s_{t+1}|s_t) \tag{2.1}$$

An example of a two-state Markov chain is given in figure 2.1. The arrows between the states indicate the transition probabilities. For example, the probability to transition from state s to state s' is 0.7, whilst the probability of staying in state s is 0.3.

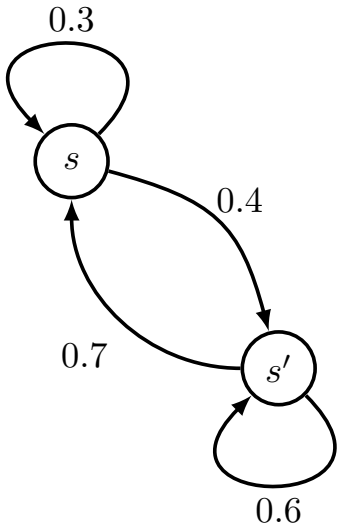


Figure 2.1: An example of a Markov chain

2.1.2 MDP

A Markov decision process is an extension of a Markov chain obtained by controlling the transition probabilities through actions. An example of a two-state, two action Markov decision process is given in figure 2.2. The dashed arrows and nodes indicate actions (there are two: a_0 and a_1). These can be used to control the transitioning of states. For example, when in state s, taking action a_1 will result in state s' with 0.9 probability (and state s with 0.1 probability).

Terminology

Another extension from Markov chains to Markov decision processes is the addition of *rewards* $R(s_t, a_t, s_{t+1})$ (wavy lines in figure 2.2). The goal of controlling the transitions in an MDP is to maximize the sum of (discounted) rewards obtained, also called the (discounted) return G:

$$G = \sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) \tag{2.2}$$

where the constant $0 \leq \gamma \leq 1$ is the so-called discount factor. This constant determines the importance of obtaining rewards sooner rather than later. For example, a γ of 1 will be indifferent on when rewards are obtained, whereas a γ close to 0 will only care about rewards obtained in the next step.

In optimizing an MDP the goal is to find a good *policy* π, which is a function $\pi(s)$ that specifies what action a to take when in state s. The optimal policy π^*

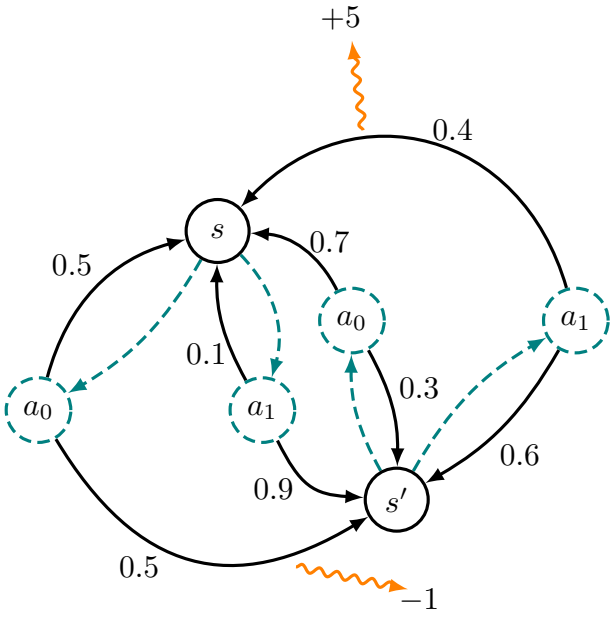


Figure 2.2: An example of a Markov Decision Process. Dashed arrows and nodes indicate actions, wavy lines indicate rewards.

is then the policy that maximizes the expected (discounted) return:

$$E[G] = E[\sum_{t=0}^{\infty} \gamma^t R(s_t, \pi^*(s_t), s_{t+1})] \tag{2.3}$$

where the expectation is taken over $s_{t+1} \sim P(s_{t+1}|s_t)$.

An example of an MDP is a game like chess. One state description s of chess is the positions of the pieces on the board (this description obeys the Markov property). However, a game like chess does not continue forever: there are termination conditions. If one of the two players checkmates the other's king, or if both players agree to a draw, the game is over. An entire game can be uniquely defined by a sequence of states, actions and rewards, this is called an *episode*. In this case the goal of the MDP becomes:

$$E[\sum_{t=0}^{T} \gamma^t R(s_t, \pi^*(s_t), s_{t+1})] \tag{2.4}$$

where $t = T$ is the time at which the termination condition is met and the episode is ended.

Generally speaking, an MDP can have a *horizon* h, which denotes the number of time-steps into the future one should plan for. Equation (2.3) shows the maximization goal of an *infinite horizon* MDP, whereas equation (2.4) is that of a *finite-horizon* MDP. Intuitively, finite-horizon MDPs are typically easier to solve than infinite horizon ones.

Formal Definition

Formally, a Markov decision process is defined by the 5-tuple (S, A, T, R, γ), where:

- S denotes a set of states called the *state space*
- A denotes a set of actions called the *action space*
- T denotes a set of conditional transition probabilities $T(s_{t+1}|s_t, a_t)$.
- R denotes a reward function $R(s_t, a_t, s_{t+1})$.
- γ denotes a discount factor, which is a constant in the range $[0, 1]$.

2.2 Partially Observable Markov Decision Process

In the previous section we learned that a Markov Decision Process (MDP) is an extension of a Markov chain. In a similar way, a Partially Observed Markov Decision Process (POMDP) is an extension of a hidden Markov model (HMM).

2.2.1 Hidden Markov Model

A hidden Markov model is a Markov chain for which the states S are hidden (or unobservable). Rather than observing the states, an HMM assumes there is another process that depends on the states S in such a way that observing this other process will allow you to learn about the states S indirectly.

An example of a two-state HMM can be found in figure 2.3. The dotted arrows and nodes indicate the observations one can receive. For instance, when in state s one can obtain observation o_0 with probability 0.1 and observation o_1 with probability 0.9.

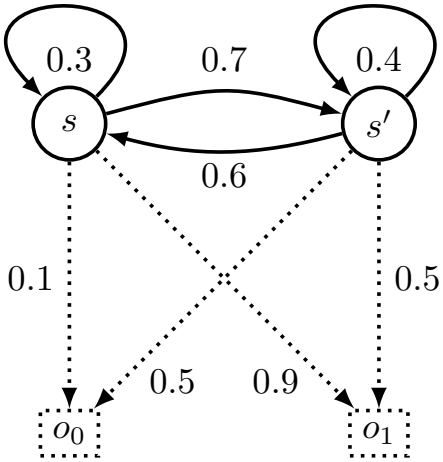


Figure 2.3: An example of a Hidden Markov Model (HMM). Dotted arrows and nodes indicate observations.

2.2.2 POMDP

A POMDP is an extension of an HMM in the exact same way an MDP is an extension of a Markov chain: with the addition of actions and rewards. A schematic view of a POMDP can be found in figure 2.4. One way of thinking of a POMDP is as an MDP for which the states are indirectly observed through some sort of sensing apparatus. For example, a POMDP can describe a robot whose position $s = (x, y)$ is its state, but whose sensors can only measure the x coordinate. Or a game of chess, for which one can only see the left half of the board. Based on knowledge of the initial state (initial position of a robot or initial state of the board in chess), the actions taken and observations received, it is possible to infer some notion (or belief) of what state the system is likely to be in.

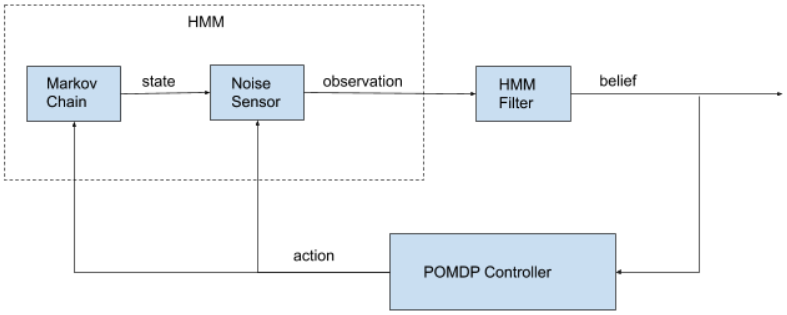


Figure 2.4: A schematic view of a POMDP.

Belief

In an MDP, one searches for a policy which is a function mapping states to actions $\pi : S \rightarrow A$. In a POMDP, the state s one is currently in is hidden. Therefore, in a POMDP, the policy is instead a function mapping a belief $b(s)$ over states to actions. Formally, the belief state b of a POMDP is defined as the posterior state distribution given the observations. It is an $|S|$ dimensional vector that lives in the $|S| - 1$ dimensional unit simplex (the belief space) $\Pi(S)$.

The belief b is a distribution over states indicating how likely it is to be in a certain state at that moment, given the entire history $\mathbb{H}$ of actions and observations up till now. Intuitively, an initial belief b_0 and the history $\mathbb{H} = \{a_0, o_0, ..., a_t, o_t\}$ is all the information one has to optimally choose a next action. However, storing the entire history is very inefficient as it is ever-growing. On the other hand, the belief b_t only requires to be maintained every time step based on the previous belief b_{t-1}, action taken a_t and observation received o_t. Given that the belief is a sufficient statistic for the history [9], it suffices to maintain the belief b_t rather than the entire history $\mathbb{H}$.

One interesting thing to note is that a POMDP can be viewed as a continuous state MDP where the MDP states are the POMDP belief states.

Formal Definition

A POMDP model can be described using the 7-tuple (S, A, T, R, Ω, O, γ) where:

- S denotes a set of states called the *state space*.
- A denotes a set of actions called the *action space*.
- T denotes a set of conditional transition probabilities $T(s_{t+1}|s_t, a_t)$.
- R denotes a reward function $R(s_t, a_t, s_{t+1})$.
- Ω denotes a set of observations called the *observation space*.
- O denotes a set of conditional observation probabilities $O(o_t|s_{t+1}, a_t)$.
- γ denotes a discount factor, which is a constant in the range $[0, 1]$.

This 7-tuple is the MDP 5-tuple with the addition of observations Ω and observation probabilities O.

2.2.3 Solution Approaches

There are many possible approaches to solving a POMDP (finding the optimal policy π^*). Different approaches are suited to overcoming different challenges

one can encounter in POMDPs. We will discuss three different dichotomies of POMDP solution approaches that will help us identify the strength and weaknesses of those approaches (more information on POMDP solution approaches can be found in appendix C). We will start by discussing the differences between exact and approximate algorithms.

Exact vs Approximate Approaches

As the name suggests, an exact approach is an approach that tries to solve the POMDP exactly. This means it will try to find the exact optimal policy that maximizes the discounted return. In an ideal scenario, we would always like to have an exact solution. However, in practice finding the exact solution might just be to difficult.

In fact, finite-horizon POMDPs are PSPACE-complete [10] and infinite horizon POMDPs are even undecidable [11]. As a consequence, even small POMDPs are often computationally intractable. The reason POMDPs are so difficult to solve exactly is due to two problems: curse of dimensionality and curse of history [8].

As mentioned before, in a POMDP we cannot observe the state directly. Instead, we have to infer a belief over the state space based on the actions taken and observations received. The space over which we need to solve the POMDP exactly is the belief space, which is the $|S| - 1$ dimensional unit simplex $\Pi(S)$. This means that the dimensionality of the problem scales exponentially with the number of states. This is referred to as the curse of dimensionality. The curse of history refers to the fact that the number of belief-contingent plans increases exponentially with the planning horizon.

This means for larger POMDPs we require approximate solution approaches. These are solution approaches that don't try to find an exact optimal policy. Instead, they search for a policy that can approximate the optimal one in terms of discounted return. Most approximate solutions overcome the curse of dimensionality by maximizing over only a part of the belief space. Similarly, they try to overcome the curse of history by considering only some belief-contingent plans.

Offline vs Online Approaches

Another dichotomy is that of offline vs online approaches. Offline and online algorithms are typically used in different settings. The offline setting is one in which all the planning/learning is done before any execution takes place. That is to say, an offline algorithm has to derive the optimal policy for every possible belief state before execution. The online setting is one in which planning/learning and execution are interleaved. Typically, in the online setting the algorithm tries to derive the optimal action to be taken in the current belief state. Once found,

this action is executed and the new belief state is observed. The algorithm then tries to find the optimal action in the new current state etc. Figure 2.5 shows the difference between the two approaches schematically.

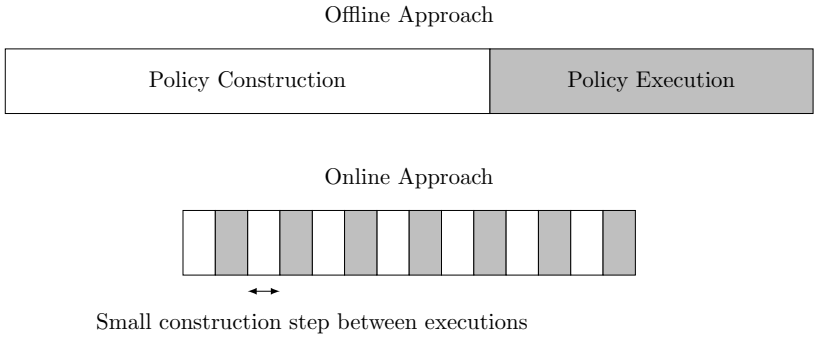


Figure 2.5: A schematic view of offline and online approaches.

There are some typical advantages and disadvantages between offline and online methods that are due to the difference in setting. As mentioned before, offline methods need to derive the optimal policy for every possible belief state before execution. For many problems, this takes too large a time to solve. Moreover, they are typically not very robust against changes in the dynamics of the environment [12]. This is because if during execution the dynamics of the environment change, the policy computed offline using the previous dynamics might not be as relevant anymore. Similarly, going from a low fidelity simulation to the real world could potentially also be thought of as a change in the dynamics of the environment (from simulation to real).

As opposed to offline methods, online methods don't have to find the optimal policy for every possible belief. They just have to find the optimal action for the current belief. This is very useful if the size of the belief space becomes very large (curse of dimensionality). Only finding the optimal policy for the current belief/state, and repeating this after every time step also makes online approaches more robust against changes in the dynamics of the environment. The same holds for uncertainty on the dynamics of the environment, a mistake in this is not propagated through more than a single action into the future.

A limiting factor of online approaches is typically that it needs to run in real-time. In other words, computation time is limited to the amount of time there is between actions. To deal with the real-time constraint, many online approaches continually keep track of the best-found solution so far. This means that they can be terminated at any time and still supply the best solution so far.

A combination of both approaches is also possible. A lot of online approaches can be used as a way to improve upon some baseline (offline) solution.

Model-based vs Model-free Approaches

The last dichotomy discussed here is that of model-based vs model-free approaches. A model-based approach is an approach that requires explicit knowledge of the POMDP model. Specifically, knowledge of the transition probabilities $T(s'|s,a)$, reward function $R(s,a)$ and/or observation probabilities $O(o|s',a)$. These functions are defined over S, A and Ω. However, these spaces can be too large to store those functions explicitly. For instance, one way to store the transition probabilities $T(s'|s,a)$ would be to have a 3-dimensional matrix over $S \times A \times S$. However, as soon as the state space gets to a certain size, storing this matrix becomes impossible. This is an issue all model-based methods share.

For larger problems (in terms of state, action and/or observation space) we require so-called *model-free* approaches. They are called model-free because they do not require explicit knowledge of the POMDP model. They typically only require implicit knowledge of the model through interaction with a generative model $\mathcal{G}$. A generative model $\mathcal{G}$ is a function which takes as input a state s_t and action a_t and generates as output the next state s_{t+1}, observation o_{t+1} and reward r_{t+1}

$$(s_{t+1}, o_{t+1}, r_{t+1}) \sim \mathcal{G}(s_t, a_t) \tag{2.5}$$

Generative models allow on-the-fly samples from the distributions $T(s'|s,a)$, $R(s,a)$ and $O(o|s',a)$ at low computational cost. This is because the complexity of sampling from a generative model is only determined by the underlying difficulty of the POMDP, not the size of the state or observation space.

2.3 Partially Observable Monte-Carlo Planning

The AML POMDP in this thesis will be relatively large and will use a model that is not 100% accurate to real life. For the reasons discussed in the previous section, this means we will look for an approximate, online and model-free approach.

The Partially Observable Monte-Carlo Planning (POMCP) algorithm is such an approximate, online, model-free approach based on the principles of Monte-Carlo Tree Search (MCTS). Before we dive into the POMCP algorithm, it is beneficial to understand the Monte-Carlo Tree Search approach for regular, fully-observable, MDPs.

2.3.1 Monte-Carlo Tree Search

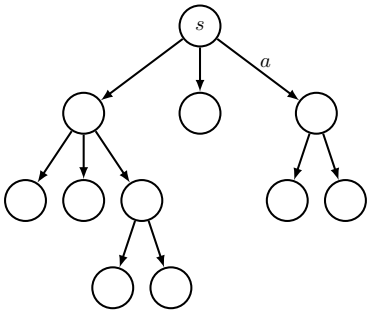
Monte-Carlo Tree Search is an online planning approach that has shown excellent performance on challenging problems [13] [14]. It is a tree search approach that evaluates states through the average outcome of simulations. It is a best-first approach that focuses on expanding the search tree in the direction that shows

the most promising results. It evaluates the leaf nodes of the tree through long-horizon simulations and is often effective without heuristics or prior knowledge. Given appropriate exploration of the search tree, it is guaranteed to converge to the optimal solution, given enough search time.

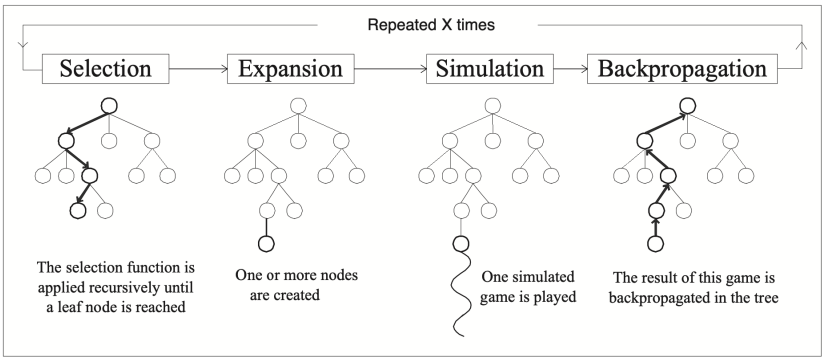
Algorithm Structure

For regular MDPs, the search tree consists of nodes representing states, and edges representing actions (see figure 2.6a). The tree is built from the root (the current state), by successively choosing edges (actions) along which to expand the tree. A Monte-Carlo Tree Search algorithm does this by iteratively applying four different phases: Selection, Expansion, Simulation and Backpropagation [14] (see figure 2.6b also from [14]). The phases consist of the following:

- **Selection** This phase consists of traversing the current search tree from the root until a leaf node is reached. The leaf node that is reached is the node that will be expanded during this iteration. The policy that is used to descent the tree is called the *tree policy*. This policy determines in what region of the search space the tree should be expanded and explored. This tree policy, therefore, has to balance between exploitation and exploration. Exploitation means that the search should be focused on regions that are the most promising, so as to not waste resources on regions that are most likely sub-optimal. Exploration means that sometimes less promising regions of the search space should be explored, to avoid overlooking the optimal regions due to some bad draws in the stochastic part of the evaluation.
- **Expansion** Once a leaf node is reached, this node is to be expanded. This means a new child node should be added and evaluated. In this way, the search tree grows by one node in every iteration.
- **Simulation** A new node (state) is evaluated by running a simulation from this state until the end of the episode (either reaching the horizon or some terminal state/condition). This simulation is generally performed using a generative model. Actions in this simulation are selected according to the so-called *rollout policy*. This policy can be completely random, or some other baseline policy on which you wish to improve. The value of the new node is equal to the return G of the simulation.
- **Backpropagation** Once the value of the new node is determined, it has to be propagated up the search tree to the root. Every node in the search tree usually keeps an average of the returns of the simulations that passed through this node. Therefore, the average value of every node in the descent path of the selection phase has to be adjusted based on the return of the new simulation.



(a) Example of a search tree in an MDP. The nodes are states and the edges are actions.



(b) The four different phases of Monte-Carlo Tree Search.

Figure 2.6: Schematic overview of Monte-Carlo Tree Search.

UCT

A crucial part of the MCTS algorithm is the tree policy. This policy determines what part of the search space to explore and therefore has to balance exploration and exploitation. Explore too much and it will take a long time to find a good policy, exploit too much and you might get stuck in a sub-optimal policy that is only optimal locally.

A widely used tree policy for MCTS is the UCT algorithm [15]. The UCT algorithm uses the UCB1 algorithm [16] to assign to every action a value:

$$u(s,a) = \mu(s,a) + c\sqrt{\frac{\log n(s)}{n(s,a)}} \tag{2.6}$$

where $\mu(s,a)$ is the average value found so far for being in node s and taking action a, $n(s)$ is the number of simulations that passed through node s, $n(s,a)$ is the number of simulations that passed through node s and edge a and c is an exploration constant. The UCT algorithm chooses the action with the highest value $u(s,a)$.

The balance between exploration and exploitation is determined through the exploration constant c. An exploration constant of $c = 0$ means no exploration takes place, a large exploration constant c means the exploration term $\sqrt{\frac{\log n(s)}{n(s,a)}}$ dominates the value $u(s,a)$ in equation (2.6). This exploration term favours actions a that have been traversed the fewest times.

Typically, nodes are initialized with $n(s) = n_{init}(s) = 0$ and $\mu(s,a) = \mu_{init}(s,a) = 0$. However, prior domain knowledge, heuristics or the solutions of an offline estimate can be used to improve the MCTS. This knowledge can be embedded in the n_{init} and μ_{init}. In this way μ_{init} is the estimated value of the prior knowledge and n_{init} is the confidence in this estimate.

2.3.2 POMCP

The POMCP algorithm is an MCTS algorithm applied to POMDPs [17]. A key difference between POMDPs and MDPs is that in POMDPs the state is not directly observed. So in a POMDP the search tree doesn't just consist of states (nodes) and actions (edges). Rather, in a POMDP we have to build a belief tree (see figure 2.7), consisting of belief nodes and action nodes (note that actions are now nodes rather than edges). Another way to view this tree is as a history-based tree. Recall from section 2.2.2 that the belief is just a sufficient statistic for the history $\mathbb{H}$. Also recall that the history $\mathbb{H}$ consists of the initial belief, together with all the actions and observations that got you up to the current point in time. So, for any belief node in the belief tree, the history is given by the descent path from the root (initial belief) to the current belief node.

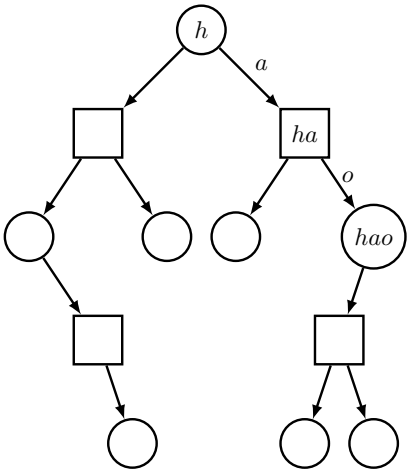


Figure 2.7: Example of a belief tree/history-based tree. The round nodes indicate histories/beliefs and the square nodes denote actions.

PO-UCT

Every belief node in the search tree represents a history h. Every action node in the tree represents a history h (of the parent belief node) followed by an action a, which we denote as ha. Just like in regular MCTS, every node keeps track of at least two variables: $\{n(h), \mu(h)\}$ (or $\{n(ha), \mu(ha)\}$). These are respectively the number of times h (or ha) was visited and the estimated value of h (or ha). The estimated value is an average of the return of all simulations that passed h (or ha). New nodes are initialized with $\{0, 0\}$ or, if there is prior knowledge, with $\{n_{init}, \mu_{init}\}$. Notation wise, a new belief h' that follows from belief h after taking action a and observing o, can be denoted with $h' = hao$.

Let's assume for now that the belief $b(h)$ is known at every belief node. In this case, to traverse the tree, starting from the root a state s is sampled from the belief $b(h)$. This state is used to sample observations, rewards and the next state from the generative model

$$(s_{t+1}, o_{t+1}, r_{t+1}) \sim \mathcal{G}(s_t, a_t) \tag{2.7}$$

In this way the tree is descended in the same way as in MCTS. What actions to choose during the descent phase is determined by the tree policy.

The tree policy for POMCP is called Partially Observable UCT (PO-UCT). As the name suggests, this is the partially observable counterpart to the tree policy in regular UCT-MCTS. Actions are selected by evaluating for every action

$$u(ha) = \mu(ha) + c\sqrt{\frac{\log n(h)}{n(ha)}} \tag{2.8}$$

after which the action with maximal $u(ha)$ is selected.

Belief State Updates

In the above, we assumed the belief $b(h)$ corresponding to every belief node h is known. For small problems, the new belief hao after action a and observation o can be derived using a belief update (calculating posterior state distribution given action and observation). However, that is a model-based approach and therefore requires explicit knowledge of the probability distributions. For problems with a large state space storing these distributions is not possible. Furthermore, even if it were possible, performing this belief update would be computationally infeasible.

So, in the POMCP algorithm, the beliefs are estimated using an unweighted particle filter over states. The reason an unweighted particle filter is used rather than a weighted one is because the former is very easy to implement using just a black box generative model. In fact, the particles in the particle filter for a node h are the states that were encountered during simulations that passed this node. So, on top of storing $n(h)$ and $\mu(h)$, belief nodes also store the states belonging to the particle filters $B(h)$ for the beliefs they represent.

Algorithm Structure

At the beginning of the POMCP algorithm, K states are sampled from the initial belief b_i. At the start of a simulation, a starting state s is sampled from these K states. The PO-UCT algorithm is used to select an action a along which to descend the tree T. The state s and action a are used to obtain a reward r, observation o and new state s' from the generative model $(s', o, r) \sim \mathcal{G}(s, a)$. The new state s' is added to the particle filter $B(hao)$. This process is repeated until a belief node is reached that wasn't encountered before. This node is initialized and the last obtained state from the generative model is added to its particle filter. This state is also used to estimate the value of this node, using the generative model and a rollout policy. Afterwards, this estimated value is propagated back through the tree T up to the root node. Pseudo-code for the algorithm can be found in algorithm 1.

This process is repeated until a termination condition is met. This concludes the planning phase of the algorithm. In the execution phase, the best-found action a is executed in the real-world, and a real observation o is obtained. The belief update for hao is now performed by defining the node hao as the new root. All other branches from h are pruned since they no longer contain reachable histories. The particle filter $B(hao)$ at hao now represents the new initial belief. If this particle filter does not yet contain K particles, new particles are added using rejection sampling. In practice, the belief update can also be combined with particle re-invigoration to avoid particle deprivation.

Algorithm 1 POMCP

procedure SEARCH(h)
  repeat
    if $h = empty$ **then**
      $s \sim b_i$
    else
      $s \sim B(h)$
    end if
    SIMULATE(s,h,0)
  until TIMEOUT
  return $\arg\max_a \mu(ha)$
end procedure

procedure ROLLOUT($s, h, depth$)
  if $\gamma^{depth} < \epsilon$ **then**
    return 0
  end if
  $a \sim \pi_{rollout}(h)$
  $(s', o, r) \sim \mathcal{G}(s, a)$
  return $r + \gamma$ ROLLOUT($s', hao, depth + 1$)
end procedure

procedure SIMULATE($s, h, depth$)
  if $\gamma^{depth} < \epsilon$ **then**
    return 0
  end if
  if $h \notin T$ **then**
    for all $a \in A$ **do**
      $T(ha) \leftarrow (n_{init}(ha), \mu_{init}(ha), \emptyset)$
    end for
    return ROLLOUT($s, h, depth$)
  end if
  $a \leftarrow \text{argmax}_b \, \mu(hb) + c\sqrt{\frac{\log n(h)}{n(hb)}}$
  $(s', o, r) \sim \mathcal{G}(s, a)$
  $R \leftarrow r + \gamma$ SIMULATE($s', hao, depth + 1$)
  $B(h) \leftarrow B(h) \cup \{s\}$
  $n(h) \leftarrow n(h) + 1$
  $n(ha) \leftarrow n(ha) + 1$
  $\mu(ha) \leftarrow \mu(ha) + \frac{R - \mu(ha)}{N(ha)}$
  return R
end procedure

2.3.3 Large Observation Space

The POMCP algorithm uses Monte-Carlo sampling of states for both the tree search and belief state updates to deal with large state spaces. Consequently, it outperformed all the previous POMDP solvers on large problems.

However, a large state space is not the only thing that complicates a POMDP solution approach. Some real world problems have not just large state spaces, but also large observation spaces. There are several extensions to the POMCP algorithm that try to solve problems with large state *and* observation spaces.

POMCP builds the belief tree by sampling observations from a generative model. When at belief node h and taking action a, every new observation o received results in a new history, resulting in a new belief node hao. The issue with large observation spaces is that the probability of obtaining the same observation in two separate simulations becomes smaller and smaller as the size of the observation space grows. In other words, if we are in h and perform a, the probability of sampling observation o in two separate simulation runs would be negligible if the observation space is large enough.

So, because each simulation will most likely sample a new observation, it will most likely create a new belief node. In this scenario our search tree would not extend deeper than the first layer (see figure 2.8 from [18]). Remember that after every creation of a new belief node, the rollout policy is applied to a state sampled from the belief. Solutions of this algorithm will closely resemble QMDP solutions. A QMDP solution assumes all partial observability disappears after the next step and will therefore never consider repeated information gathering steps.

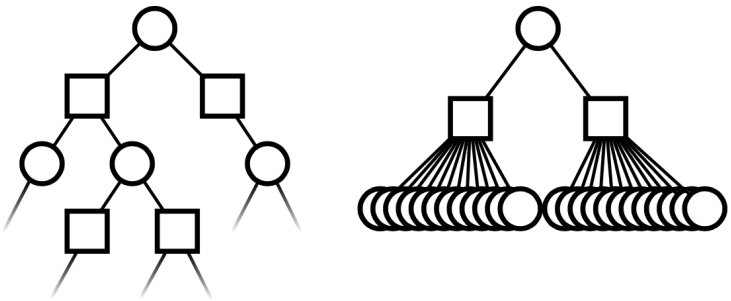


Figure 2.8: Example showing that for large observation spaces the search tree never extends beyond the first layer.

Furthermore, if we want the POMCP algorithm to improve upon some baseline solution (perhaps found offline), this can be achieved by using this baseline

as the rollout policy. The issue then becomes that, because the tree will never extend deeper than the first action, this method will never discover improvements on the baseline that differ from it in more than one action.

We refer to appendix D for a more in-depth summary of the POMCP extensions that try to deal with large observation spaces.

Chapter 3

Combinatorial Action Spaces

In the previous chapter, we introduced the POMCP algorithm as an MCTS based approach for partially observed environments that handles large (even continuous) state spaces using Monte-Carlo sampling. We also briefly discussed some of the challenges of dealing with large observation spaces. However, this chapter is about the challenges of a large action space. In particular, it summarizes some of the concepts used in this thesis to exploit the combinatorial structure of the AML action space.

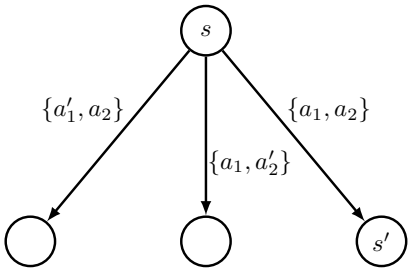
Large action spaces are an issue because the size of the action space determines the branching factor of the search tree (the same as with a large observation space). This means large action spaces will result in broad, shallow search trees.

There exists some research on POMCP-based methods that deal with continuous action spaces [18]. They use *progressive widening* of the search tree to limit how broad it can be. However, this is a rather crude solution that ignores parts of the action space if it gets too large, which severely limits the possibility of exploration. In the continuous case, ignoring parts of your search space is inevitable due to it being infinitely large.

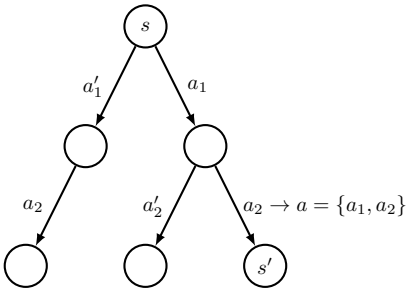
In the discrete case, however, there can be more sophisticated solutions. Sections 3.1 and 3.2 respectively discuss the concepts of action chains and Directed Acyclic Graphs (DAG) in the search tree. Some of the methods proposed in chapter 5 build DAGs into their search trees. Section 3.2.1 therefore discusses the Upper Confidence bound for Directed acyclic graphs (UCD) framework, which is an adaption of MCTS to search DAGs. Related to UCD is the *update-all* algorithm which is discussed in section 3.2.2.

3.1 Action Chains

In some POMDP domains, it is possible to leverage the structure of the actions to reduce the complexity of the action space. For instance, in multi-agent POMDPs the joint action of multiple agents can be decomposed into sub-actions performed



(a) An example of a partial search tree. The rightmost edge denotes joint action $a = \{a_1, a_2\}$.



(b) An example of a partial search tree with action chains. The rightmost path denotes selecting first sub-action a_1 and then a_2, resulting in action $a = \{a_1, a_2\}$.

Figure 3.1: Example of a regular partial search tree and one with action chains for $N = 2$ and $K = 2$.

by individual agents. In this way, the joint action can be decomposed into an *action chain* [19] where each agent decides on its action sequentially (instead of all of them at once).

The benefit of this approach is that value can be assigned to intermediate steps in the action chain. For example, suppose we have a multi-agent problem with $K = 10$ agents, each with an individual action space of size $N = |A_i|$. An action chain would now consist of a tree with $|A_i|$ actions on the i'th level (representing the possible actions for agent i) (see figure 3.1). With an action chain, it is possible to assign an intermediary value to agent 1 choosing action a_1, rather than only assign values to the joint actions of all the agents combined (the leaf nodes in the action chain). If the actions chosen are sufficiently independent, this could significantly help in guiding the search through the exponential joint action space (consisting of the Cartesian product of A_i).

3.1.1 Selecting Nodes From a Graph

In [20], the authors present a software agent that recommends sequential intervention plans to raise awareness about HIV among homeless youth. The fundamental goal of this paper is to choose individuals from a social network in such a way to maximize influence spread across the network. The actions are defined by sequentially selecting a fixed number of nodes (individuals) from a graph (social network). Suppose $K = 6$ nodes could be picked every round (the group size for an intervention) out of a $N = 150$ node graph, the action space consists of every possible way to pick 6 nodes out of 150 (order independent): $|A| = \binom{150}{6} \approx 14 \times 10^9$. The main limitation of the POMCP method from section 2.3 is the memory needed to store this (very broad) search tree.

Due to this limitation, the authors in [20] limit themselves to only performing a search over the next action to take (so a search tree that only goes down 1 level). However, in order to build a search tree that covers all possible next actions to take, you would still need to store over 14 billion nodes in memory (for the example above), this isn't feasible. Instead, they build a K-level tree (where K is the number of nodes picked per round), which is an action chain tree consisting of only the first combinatorial action.

3.2 Directed Acyclic Graphs

In certain problem domains, it is possible to create a search Directed Acyclic Graph (DAG), rather than a search tree, through the application of transpositions [21]. For instance, in chess, a transposition occurs when a sequence of moves results in a position that is also reached by another sequence of moves. The position of the pieces of the chessboard describes a state space that has the Markov property. In other words, what sequence of moves gets you to a certain position has no influence on the future of the game, the only thing that matters is the current position of the pieces.

The value function only depends on the current board position. If two sequences of moves lead to the same board position, they would have the same value going forwards. In general, a transposition occurs in any MDP when a sequence of actions leads to the same state as another sequence of actions. Linking the two action sequences of a transposition in the search tree means there is no longer a single unique path from the root to any leaf node. Therefore, the search tree is no longer a tree, but rather a search DAG (see figure 3.2).

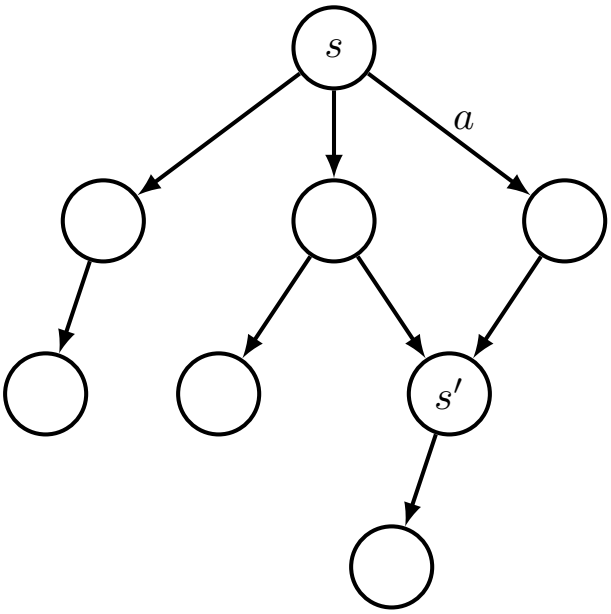


Figure 3.2: An example of a search Directed Acyclic Graph (DAG). In this example there are multiple sequences of actions that lead to state s'.

3.2.1 Upper Confidence Bound for Directed Acyclic Graphs

In [22] they propose a framework to deal with transpositions in MCTS called Upper Confidence bound for Direct acyclic graph (UCD). In their framework, descend through the game tree works as follows: At a node, an action is chosen through some selection procedure. If the edge associated with this action already exists, move along this edge to another node and repeat. If the edge does not exist, consider the newly achieved state. This new state might already be associated with a node in the tree. If so, connect it to this node and repeat. If not, create a new node and apply the rollout policy.

Backpropagation

In terms of backpropagation, the UCD framework applies the same backpropagation strategy as regular MCTS as if there were no transpositions. This strategy is to update only the descent path, i.e. the path through which the leaf node was reached during the selection phase of the current MCTS iteration. The problem is now that not all information useful in the selection phase is stored locally (see figure 3.3 from [22]). In order to make optimal decisions, it could be necessary to look ahead further down the DAG. If a transposition occurs further down the search DAG, that information is only available at the location of the transposition. So in order to take into account information from transpositions, the tree

policy will have to look further down the tree before it makes a decision.

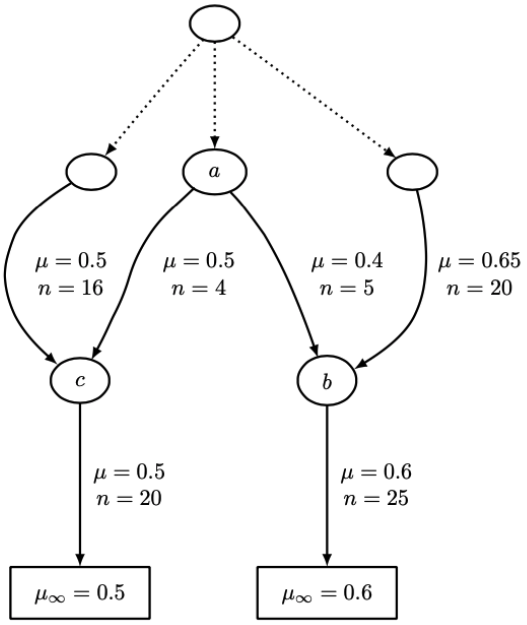


Figure 3.3: An example showing that by updating only the descent path not all relevant information is stored locally. At node a, choosing optimally between c and b is not possible with only the local information.

Selection

The effects of a transposition can be found by moving sufficiently far down to the tree until the transposition node is encountered. In order to use this for the tree policy, the UCD algorithm adapts the regular UCT variables (value and count) into versions that aggregate the values going down the DAG for a certain depth d. The first variable they define is the *adapted score* μ_d:

$$\mu_0(e) = \mu(e) \tag{3.1}$$

$$\mu_d(e) = \frac{\mu'(e) \times n'(e) + \sum_{f \in c(e)} \mu_{d-1}(f) \times n(f)}{n'(e) + \sum_{f \in c(e)} n(f)} \tag{3.2}$$

where $\mu(e)$ is the UCT value function for edge e, $c(e)$ are the children of edge e, $n(f)$ is the UCT count and μ' and n' are the initial values for μ and n respectively. The other variable is the count variable n which only counts the actual amount of times a node is passed during the selection procedure.

They also define an *adapted count* n_d:

$$n_0(e) = n(e) \tag{3.3}$$

$$n_d(e) = n'(e) + \sum_{f \in c(e)} n_{d-1}(f) \tag{3.4}$$

Note that in a way the larger the parameter d is, the more transpositional information is taken into account. If d is very small for instance, only transpositions that occur close to the current node are considered. Whereas if d were really large (or infinite), all the transpositions that will ever occur in the future are considered.

Both the adapted variables combine in the definition of the UCD tree policy:

$$u_{d_1,d_2,d_3} = \mu_{d_1}(e) + c\sqrt{\frac{\log(\sum_{f \in b(e)} n_{d_2}(f))}{n_{d_3}(e)}} \tag{3.5}$$

where $b(e)$ are the sibling edges of e including e. Note that the UCD tree policy has three depth parameters which allow for variability in how much different parts of the tree policy take into account transpositional information.

Notes on implementation

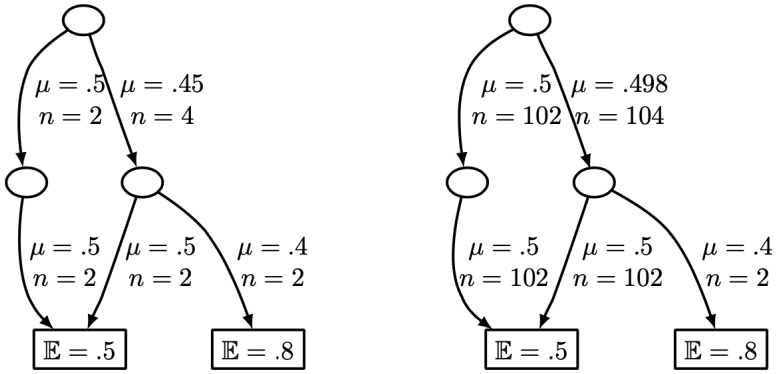
Instead of descending the tree up to depth d_i for every node in the selection procedure for all three parts of the tree policy, it is possible to define a procedure to store the adapted variables on the edges and update them during the backpropagation phase.

3.2.2 Update-All

In addition to introducing the UCD framework, the authors of [22] also briefly mention the update-all algorithm. The update-all algorithm derives its name from its backpropagation strategy. This strategy is as follows: rather than updating only the descent path, update all ancestor paths as if they were the descent path (taking care to update every ancestor only once). During the selection phase, the update-all algorithm then applies the UCT algorithm for its tree policy. In other words, the update-all algorithm is the same as plain MCTS, but with a backpropagation phase adjusted to search DAGs that updates all paths from the leaf node to the root.

The update-all algorithm is designed to maximally share the information obtained from a playout. It does so by backpropagating this information equally along every possible ancestor path. In this way, it shares this information with every part of the search tree for which this information is thought to be relevant (given the current known search DAG).

It is desirable to maximally use the available information to be able to make the best informed decisions. However, the authors of [22] opt for the UCD framework rather than the update-all algorithm. They show that the update-all algorithm can perform sub-optimally in a certain scenario. This is demonstrated with a counter-example shown in figure 3.4 (from [22]). In this counter-example, the optimal action on the lower right is masked by an unlucky sample rollout. This poor estimate is never adjusted due to the update-all strategy always updating the upper-right edge, even though the UCT algorithm always chooses the left-most descent path.



(a) Initial settings (b) 100 playouts later

Figure 3.4: A counter-example for the update-all backpropagation procedure.

Part III

Anti-Money Laundering

Chapter 4

POMDP Specification

This chapter will specify the Anti-Money Laundering (AML) problem in terms of the POMDP framework discussed in chapter 2. The formal definition of the AML POMDP is specified in section 4.1. Afterwards, we specify the transition and observational model further in section 4.2.

Informally, the AML process could be defined as follows:

- **State** s: The transactional graph and any other knowledge we might have on the entities in it.
- **Action** a: A subset of the nodes in the transactional graph of a particular size (equal to the number of investigators).
- **Reward** r: A positive value whenever a SAR is filed.
- **Observation** o: Any information we obtain from an investigation.

This informal definition helps in conceptualizing the more complicated parts of the formal definition stated in the next section.

4.1 Formal Definition

Before we define the AML process, we should specify that AML investigations in real life are a very complicated process. In order to gain some traction on solving the decision making problem in this thesis, we will make a number of simplifying assumptions.

Assumptions

The first simplification is that we assume the transactional graph to be static (for instance a snapshot during a certain time window). This means we will not consider new accounts or transactions.

Another simplification is that we assume an immediate reward feedback. In reality, there might be a delay between the finishing of an investigation and the submission of a SAR to the public prosecution service. In our model, we assume the filing of a SAR is immediate and therefore a positive reward can be given immediately after an investigation has concluded.

Additionally, a POMDP assumes a discrete time-step process. This does not necessarily apply to the AML process, however. In real life, investigations can take a variable amount of time depending on several things, including external resources and whether a node is actually fraudulent or not. This could potentially be modelled using a fixed time-step denoting a fixed unit of investigation time, together with a variable action budget (the number of nodes that can be selected for an action). In this thesis, we simplify this process by assuming instead a fixed size time-step as well as a fixed size action budget.

4.1.1 Definition

With the above assumptions and informal definition in mind, the formal POMDP definition of the AML process used in this thesis will be the following:

- **State space** S: Our state is a transactional graph $G = (V, E)$. The nodes V denote the accounts and the edges E are the transactions. All nodes $v \in V$ have a set of binary labels $L(v) = \{l^r(v), l^o(v)\}$ and a set of features $F(v) = \{f^r(v), f^o(v), f^c(v)\}$. The labels $L(v)$ indicate whether the node v is illicit or not: l^r is the real status of the node and l^o the observed one (determined through investigation). The features are divided in a similar way: f^r is the set of features with their real values and f^o is the same set but with the values that are being observed (also determined through investigation). The set f^c is a set of binary labels indicating for every feature whether we believe them to be correct or not. So, at time t the state s_t consists of the tuple:

  $$s_t = \{G, L_t, F_t\} = \{G, l^r, l^o_t, f^r, f^o_t, f^c_t\}$$

  where we use notation: $F_t = \{F_t(v)\}_{v \in V}$. Note that G, l^r and f^r are time independent since we assume a static transactional graph.

- **Action space** A: The action space consists of all possible subsets of nodes V of a fixed size K (the action budget):

  $$A = \{a \subset V \quad | \quad |a| = K\}$$

  The nodes in a are the ones to be investigated.

- **Transition probabilities** T: The distribution that describes how our state transitions from one time step to another. Since we assume a static transactional graph, this only includes the dynamics of l_t^o, f_t^o and f_t^c. How the state changes depends on the previous state and the action taken: $T(s_{t+1}|s_t, a_t)$.

- **Reward function** R: The reward is a positive constant $r = 1$ every time a node gets labeled as illicit.

- **Observation space** Ω: An observation o_{t+1} consists of the observed labels $l_{t+1}^o(v)$ for every node v that is investigated (which includes at the very least the nodes in the action a, but might include more).

$$o_{t+1} = l_{t+1}^o(v) \text{ for } v \in \text{ set of investigated nodes}$$

- **Observation probabilities** O: The observation probabilities are the probabilities of observing o_{t+1} given s_{t+1} and a_t. Since $o_{t+1} = l_{t+1}^o$ is also part of the state space this can be defined in terms of the transition probability:

$$O(o_{t+1}|s_{t+1}, a_t) = O(l_{t+1}^o|s_{t+1}, a_t) = T(l_{t+1}^o|s_t, a_t)$$

In a POMDP the observational probabilities depend only on the action taken and the state transitioned to based on that action: $O(o_{t+1}|s_{t+1}, a_t)$. This is why, besides the transactional graph and its features, we added to the state space all the relevant information needed for modelling the investigations (things like l_t^o, f_t^o and/or f_t^c).

The observation space

There are some things to note about the observation space in the formal POMDP definition above. In the informal definition, the observation space includes every relevant piece of information obtained from the investigations. This should therefore probably include the observed feature values f^o and whether those values are believed to be correct f^c (both potentially observed during an investigation). However, this would increase the size of the observation space significantly. This would be accompanied with all the challenges of large observation spaces described in section 2.3.3.

However, during our experiments, we discovered that our generative model does not heavily rely on the specifics of the information in f^o and f^c. Therefore, when using that model, we can significantly reduce our observation space by only including l^o (which is very relevant for our model). In this way, we are ignoring part of the information we could potentially use (by excluding f^o and f^c from the observation space), in favour of reducing the search space complexity (smaller observation space). If in the future the generative model changes, it might become more relevant to include f^o and f^c in the observation space.

4.1.2 Large Scale

The AML POMDP defined in this chapter has a relatively large scale two respects: the state space and the action space.

The state space

The state of the AML POMDP defined above is the transactional graph together with labels and numerical features. The state space is therefore the space of all possible transactional graphs together with the spaces for all the possible feature/label values. The space of all possible transactional graphs is unbounded (you can always add another transaction to obtain a new valid transactional graph). When combined with the continuous space of the numerical features it becomes clear that this state space is relatively large.

The action space

The POMDP defined above also has a large (combinatorial) action space. This is due to the fact that our action consists of selecting a subset of nodes $a \subset V$ of size K (our action budget) out of all the possible nodes V in the transaction graph. In other words, our action is defined as selecting a subset of a particular size out of a larger set of possibilities, resulting in an action space of size $|A| = \binom{|V|}{K}$.

4.2 Model Specification

Because the state space of the AML problem is so large, it will be impossible to explicitly model every single transition probability $T(s_{t+1}|s_t, a_t)$ for every possible pair of (s_{t+1}, s_t). Instead, we will define a generative model $\mathcal{G}$ that generates the next state s_{t+1}, observation o_{t+1} and reward r_{t+1} given input s_t and a_t:

$$(s_{t+1}, o_{t+1}, r_{t+1}) \sim \mathcal{G}(s_t, a_t) \tag{4.1}$$

This generative model is also what we will require in order to perform Monte-Carlo sampling in our POMCP-based solution approaches.

There exists no precedent for research into AML investigations and therefore we will have to define and implement this model ourselves. The real process is of course very complicated, involving human judgement and external resources. In this thesis, we will use a simplified model that hopefully captures the relevant elements of the real-world process.

4.2.1 Basic Structure

Before we discuss the specifics of our transition probabilities T, we will discuss the basic structure of our model. Our model will take as input a current state s_t and an action a_t. The action can consist of multiple nodes $a^i \in a_t$ that need to be investigated. The idea behind this action space is that the number of nodes to be investigated equals the number of human investigators available. The investigation process roughly consists of the following steps:

1. Observe some information (f^o, f^c) on the features of the node a^i and some of its neighbours.
2. Repeat the above for all $a^i \in a_t$
3. Based on the information gathered (f^o, f^c) during the investigations, label all investigated nodes (l^o) licit or illicit with a certain probability.

In the first step, we try to model real-life by investigating more than just the node a^i under formal investigation. Part of the investigation into a^i is investigations into suspicious activity of a^i's network. In reality, the (partial) investigation into a suspicious neighbour of a^i could reduce the investigation time needed for this neighbour when it is under formal investigation itself.

However, we assumed a fixed size time-step in our model. This behaviour is a potentially a large component of the dynamics of the investigation process. So, in an attempt to fit this behaviour in our simplified model, we will add the investigation into a^i's suspicious neighbour together with its labeling (and therefore submission of a SAR) to the current time-step in which a^i was investigated.

Parameters

How many neighbours and what degree of neighbours of a^i are to be investigated are problem-specific parameters of our model. These parameters are necessary to be able to adapt the scope of the investigations based on the problem size. The parameters will be referred to as the following: the investigation budget K_I and the investigation depth d_I.

The investigation budget K_I determines how many neighbours (besides a^i) will be considered for investigation. For instance, an investigation budget of 3 means a^i and two of its neighbours will be investigated. The investigation depth d_I determines the degree of neighbours under consideration. An investigation depth of 2 means all 1-hop and 2-hop neighbours of a^i will be considered.

4.2.2 Transition Probabilities

In our model, the only parts of the state space $s_t = \{G, L_t, F_t\}$ that are dependent on time are (l_t^o, f_t^o, f_t^c). The transition model $T(s_{t+1}|s_t, a_t)$ therefore can be thought of as

$$T(s_{t+1}|s_t, a_t) = T(l_{t+1}^o, f_{t+1}^o, f_{t+1}^c|s_t, a_t) \tag{4.2}$$

The features that are believed to be correct f_{t+1}^c follow directly from the previous f_t^c and the new feature values observed in the current iteration. This is because when a new feature value is observed, they are also assumed to be correct. So, f_{t+1}^c is f_t^c plus all newly observed features. This set $f_{t+1}^c(v)$ will be used as a measure for how much knowledge is available on a particular node v at time $t + 1$.

Observing the values of features f_{t+1}^o is modelled through a sequence of n independent Bernoulli trials. The outcome 'success' is obtained with probability p and denotes the case in which this value is observed. The 'failure' outcome denotes the case in which this value is not observed. The number of Bernoulli trials n is equivalent to the total amount of features. In other words, during an investigation process for a node v, a coin is flipped for every feature in $F(v)$ to determine whether this feature value is observed or not. Through observing new feature values, we can increase the amount of features assumed to be correct (in f_{t+1}^c), which increases our measure for how much knowledge is acquired for a particular node.

A node v is labeled ($l_{t+1}^o(v)$) licit or illicit based on three different metrics. The first one is the amount of knowledge available on node v (for which we use $f_{t+1}^c(v)$ as a measure). The second one is the amount of knowledge available in the local network around v (for which we use f_{t+1}^c of the neighbours). Lastly is the amount of neighbouring nodes that was already found to be fraudulent in previous investigations (for which we use l_t^o of the neighbours).

Note that one further simplification made here is that labeling multiple nodes in a single time-step is done independently. In other words, labeling a node is independent of the results for other nodes investigated in the current time-step. If this were not the case the labeling of a node v would have to somehow depend on the labels l^o of its neighbours at time $t + 1$.

4.2.3 Initial Conditions

Above, we described how we transition from a certain state to the next. We will now describe how we generate an initial state s_0. Just as stated above, the only dynamic parts of our state space are (l_t^o, f_t^o, f_t^c). We will start by describing how to generate the initial values for those variables.

Dynamic Variables

The simplest variables to initialize are the labels l_0^o. We assume that at the start of the episode we haven't found any of the illicit nodes yet. So we initialize all l_0^os as licit.

The observed feature values f_0^o will be initialized by taking the real values f^r, and adding noise to some of the values. We decide what value to add noise to by a sequence of independent Bernoulli trials (in the same way as for observing feature values described above). This is supposed to simulate that in real life most of the features in the transactional network are generated through some automatic or machine learning methods. These methods are never fully error proof and therefore it wouldn't be realistic for the observed values f_0^o to be the same as the real ones f^r.

Lastly, the values f_0^c denote what feature values f_0^o we believe to be correct. We can make an initial guess on what values we deem correct based on our prior knowledge of the process that derives these values. In our model, this process is a series of Bernoulli trials. Therefore, we will make an initial guess for what values are correct f_0^c through a series of Bernoulli trials in the same way as described above for f_0^o. If in the future the process for initializing f_0^o becomes more complicated, the process for initializing f_0^c can change in the same way.

Static Variables

In the previous section, we described how we initialize the dynamic parts of our state (l_t^o, f_t^o, f_t^c). The static parts of our state (G, l^r, f^r) are assumed to be obtained directly from our transaction network. For the empirical evaluation, we will need to acquire a real/generate a synthetic transaction network for this purpose.

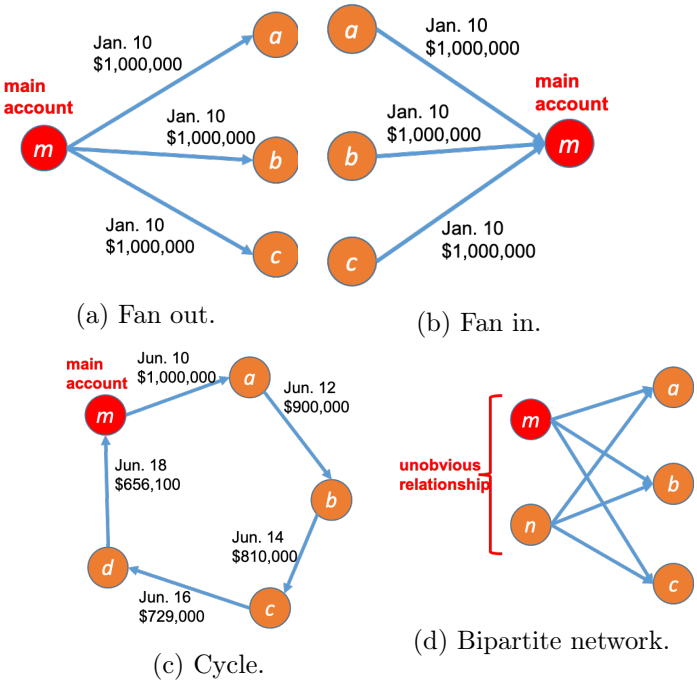
One candidate for a real transaction network could be the Elliptic Data Set [3] for cryptocurrency. This dataset contains real Bitcoin transactions with real labels obtained from uncovered scams, malware, etc. The issue with this dataset is the potential mismatch between the graph structure of this cryptocurrency and normal financial transactions. In the Elliptic Data Set the nodes represent Bitcoin transactions and the edges represent the flow of Bitcoins. We found in our experiments that the model we created for the investigations described in this chapter did not properly match with the structure of these graphs. Therefore, for the empirical evaluation in this thesis, we opt for using the AMLSim simulator.

AMLSim

The AMLSim simulator [4] is designed to generate a series of banking transaction data together with a set of known money laundering patterns. The simulator

works by first generating a transaction network structure that includes some suspicious transaction patterns such as cycle, fan-in/out and bipartite network with non-obvious relationships (see figure 4.1). It then performs an actual transaction simulation on top of this structure. A schematic for this procedure can be found in figure 4.2.

The AMLSim simulator labels all accounts involved in the AML typologies as illicit. In this way, the graph G and the true labels l^r can be straightforwardly obtained from the AMLSim transaction network. Unfortunately, unlike the Elliptic Data Set, the AMLSim simulator does not generate node features f^r that can be useful predictors for fraud used in the risk scoring.



(a) Fan out.

(b) Fan in.

(c) Cycle.

(d) Bipartite network.

Figure 4.1: Examples of the AML typologies used in AMLSim. Figure from [23].

However, we overcome this by creating a 'perfect' risk score that is 1 if the node's true label l^r is illicit, and 0 otherwise. Because the risk score is now independent of the feature values f^r and f_t^o, we create dummy variables for these features. Our investigations only directly depend on the variables f_t^c, which can still be initialized and updated in the exact same way described above, regardless of the variables f^r and f_t^o being real or dummies.

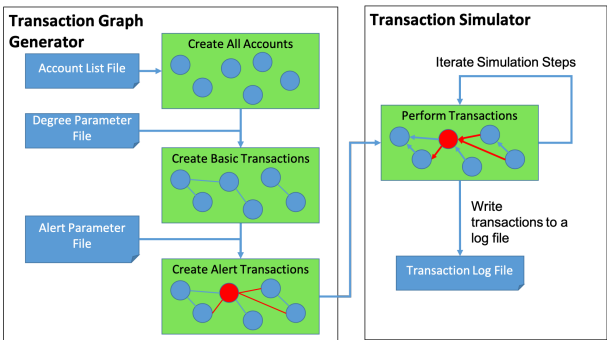


Figure 4.2: Workflow of the AMLSim simulator. Figure from [23].

Chapter 5

Solution Approaches

This chapter will introduce the solution approaches proposed in this thesis to optimize the AML POMDP defined in chapter 4. Several POMCP-based methods are discussed together with their potential limitations and the motivation for using them in this thesis. We will first propose the POMCP method with a memory-efficient implementation in section 5.1. Afterwards, we introduce the concept of action chains to the POMCP method and define the POMCP-AC algorithm in section 5.2. Lastly, we introduce the concept of transpositions to the action chains of the POMCP-AC method in section 5.3, creating directed acyclic graphs in the belief tree. In that section, we discuss two POMCP-based methods: POMCP-UCD and POMCP-UA, based on the UCD and update-all algorithms respectively.

5.1 Memory-Efficient POMCP

As discussed in section 2.2.3, a large state space means it is infeasible to use a model-based approach that requires explicit knowledge of the model's probability distributions. Additionally, the size of the state space determines the dimensionality of the belief space. Therefore, due to the curse of dimensionality, an offline approach would not be tractable. Finally, for the same reason of scale, an exact solution will be infeasible. As discussed in the previous chapter, the AML POMDP has a relatively large state space. This means we require a model-free, online and approximate solution approach. One such approach is the Partially Observable Monte-Carlo Planning (POMCP) algorithm.

The POMCP algorithm does not use the structure of the action space in guiding its search through the search tree. For example, in the POMCP algorithm the joint action $\{a, b\}$ and the joint action $\{a, c\}$ are not related in any way. Even though, with our knowledge of the action space structure, we know they are related in that they both contain node a.

However, all the other methods in this thesis are extensions of the POMCP algorithm designed to exploit the structure of the combinatorial action space.

The POMCP method discussed in this section is, therefore, a useful baseline to compare the other methods with. Comparing the other methods to the POMCP method will serve as an ablation study to examine the benefits of exploiting this action space structure.

Memory efficiency

The AML POMDP does not only have a large state space, but it also has a large action space. The regular implementation of the POMCP algorithm from section 2.3 initializes every possible action node (of which there are $|A|$) whenever a new observation node is created (and then performs a rollout). If the action space is large, this will result in an incredibly large amount of nodes initialized in memory that store no new information (just the initial values). This is not very memory-efficient. In fact, this implementation of the algorithm cannot even solve small combinatorial problems without running into significant memory issues due to the large action space.

So, a memory-efficient implementation of the POMCP algorithm would be one in which action nodes are only created once they are actually traversed during the selection phase. These nodes will hold new information due to the backpropagation of the results obtained from the rollout. In order to perform the selection phase just like the regular POMCP implementation, the memory-efficient one will have to temporarily initialize the relevant and currently uninitialized action nodes with the initial values. At most this implementation will have to temporarily initialize $|A|$ nodes at a time. This is potentially a lot less than the $|A|$ nodes the regular implementation initializes every time a new observation node is created.

Time consumption

It is worth noting that the memory-efficient implementation has a much smaller memory footprint, but will be more time-consuming. This is because it needs to initialize actions not just once, but every time these actions are considered by the tree policy during the selection phase. In this sense, this implementation is a memory versus time trade-off.

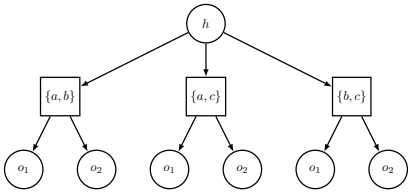
If the initial values are constants (for instance $n_{init} = 0$ and $v_{init} = 0$) or based on some simple heuristic, this trade-off will most likely be worth it. However, if the initial values contain prior knowledge obtained through some complex offline solution, the added time complexity might make the memory-efficient implementation unusable.

5.2 Action Chains

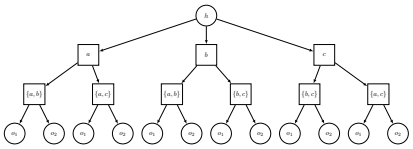
In the paper on sequential intervention plans for homeless shelters [20], they deal with the combinatorial action space by searching over a K-level tree instead of a regular belief tree. This K-level tree can be viewed as an action chain for the next possible joint action. However, performing a search in only the next action is like building a search tree that is only a single layer deep. This has all the limitations discussed in the sections on large observation spaces 2.3.3 (it converges to the QMDP solution). Preferably, we would like to build a search tree that goes beyond the single next action.

5.2.1 POMCP-AC

We propose to use the concept of action chains in the POMCP method in an algorithm we call POMCP with Action Chains (POMCP-AC). Recall that the POMCP algorithm builds a belief tree with action nodes and observation nodes (see figure 5.1a). We can introduce the concept of action chains to belief trees by building the action chains instead of action nodes in between the observation nodes. Figure 5.1 shows the difference between the two types of belief trees.



(a) An example of a regular belief tree.



(b) An example of a belief tree with action chain.

Figure 5.1: Examples of a regular belief tree and one with action chains for $N = 3$ and $K = 2$.

We include this method because the two methods proposed in the next section extend the POMCP-AC method by adding the concept of transpositions. Comparing those methods with the POMCP-AC method will therefore act as an ablation study for the addition of transpositions.

Furthermore, the K-level tree used in [20] closely resembles the POMCP-AC method. Comparing the other methods in this thesis with POMCP-AC can therefore act as a justification to apply those methods to the problem domain in [20].

The POMCP-AC algorithm can be applied to any domain with an action space in which it would be meaningful to assign value to sub-actions. The key idea is that this method can improve upon POMCP by exploiting the structure

of the action space to guide the tree search in the direction of the most promising regions.

Symmetries

In an action chain, there are $N-m$ children for every node in the m^{th} level of the action chain (where N is the number of nodes in the graph). This means, for K number of nodes picked per round, the action chain has $\prod_{m=0}^{K-1}(N-m) = \frac{N!}{(N-K)!}$ possible leaf nodes (actions). This is more than the original $\binom{N}{K} = \frac{N!}{(N-K)!K!}$ in the combinatorial action space. This is because in the action chain, selecting a node a first and then a node b, leads to a different part of the tree than selecting node b first and then a, even though in the combinatorial action space the action $\{a, b\}$ is the same as $\{b, a\}$. In other words, the action chain ignores the symmetry of actions in the combinatorial action spaces.

Because the POMCP-AC method ignores the symmetry of the combinatorial action space, we do not necessarily expect it to always outperform the other methods in this thesis. In fact, it might even be outperformed by the POMCP method in some scenarios, since this method implicitly takes into account the combinatorial symmetry in the definition of its joint action space.

5.3 Directed Acyclic Graphs

In problem domains where the order of (sub-)actions doesn't matter (such as a combinatorial action space), we would like to have the path $a \rightarrow b$ lead to the same part of the search space as path $b \rightarrow a$. One way to achieve this is to connect the path $a \rightarrow b$ to the path $b \rightarrow a$ in the search tree (by letting them point to the same node). In this case, the search tree is no longer a tree, but rather a Directed Acyclic Graph (DAG).

We propose to extend the POMCP-AC method of the previous section by adding transpositions to the action chains. A transposition in this context connects two paths in the action chain of a belief tree that lead to equivalent joint (sub-)actions. This adapts the POMCP-AC method into a method that can exploit the symmetries of the combinatorial action space.

Due to the transpositions, the belief tree transforms into a belief DAG (see figure 5.2). Note that the transpositions only occur within the action chains of the belief tree. The belief DAG can therefore be viewed as a belief tree consisting of smaller action-DAGs at the level of the action chains, rather than a single large DAG.

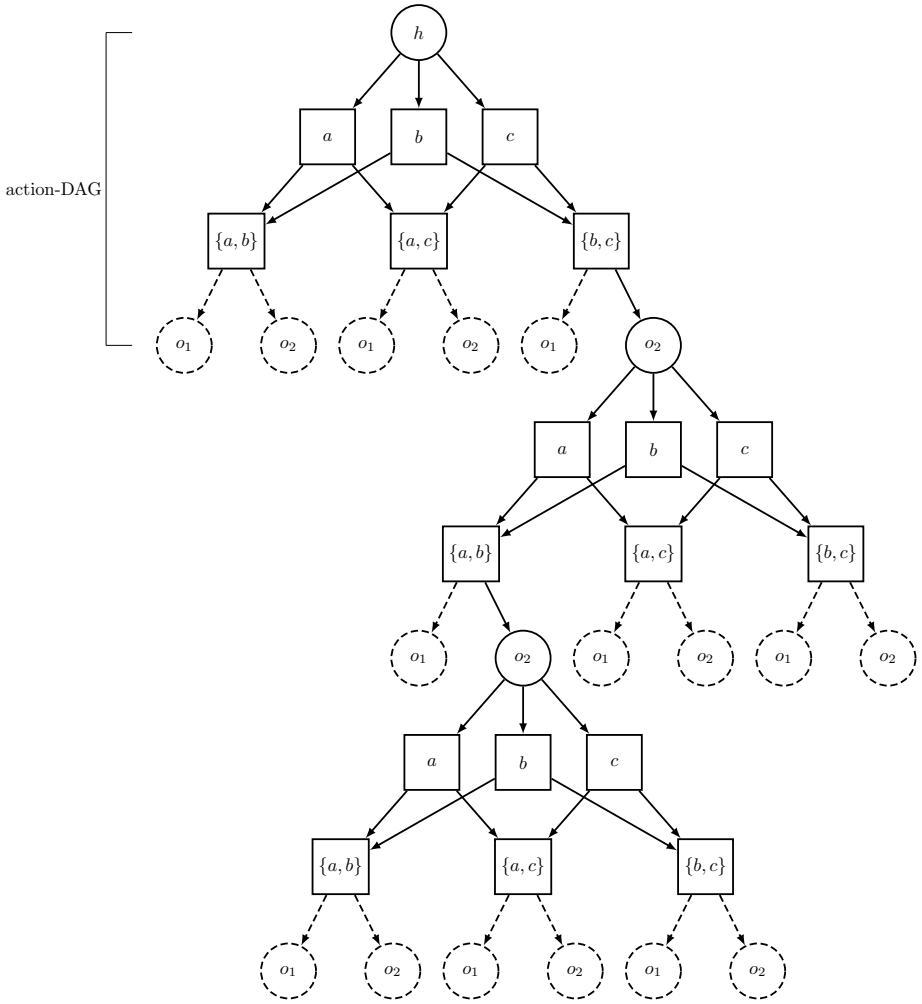


Figure 5.2: An example of a (partial) belief Directed Acyclic Graph (DAG) for $N = 3$ and $K = 2$. The part in between any two observation nodes is referred to as the action-DAG.

In this section, we will introduce two methods that are designed to deal with a belief DAG rather than a belief tree: POMCP-UCD and POMCP-UA. The POMCP-UCD algorithm applies the UCD framework for MCTS to the action-DAGs of the belief DAG. The POMCP-UA algorithm is similar to POMCP-UCD but applies the Update-All (UA) algorithm instead of UCD.

Exploiting prior knowledge

The UCD framework was designed with general problem domains in mind. In general, one does not know about the existence of a transposition until it is encountered during the traversing of the search tree. In other words, transpositions

are initialized in hindsight: we initialize a transposition after we realize the state we just traversed to already exists in the search tree.

In the combinatorial action space, transpositions are due to the symmetries of the combinatorial structure. These symmetries are known before encountering them in the search tree and are therefore a form of prior knowledge. This prior knowledge can be exploited in various degrees by initializing transpositions into the belief DAG before encountering them. On one hand, you could not exploit it at all by ignoring the prior knowledge and initializing transpositions only when encountering them (like the UCD framework). On the other hand, you could exploit the prior knowledge fully by initializing the entire action-DAG before traversing it (since we have prior knowledge of every possible transposition). There exists an inherent trade-off between the two as exploiting the prior knowledge more will come at an increased cost in memory and time consumption.

The POMCP-UCD and POMCP-UA methods not only differ in how they traverse and backpropagate through the action-DAG (UCD vs update-all). They also differ in what degree they exploit the prior knowledge of the combinatorial action space structure.

5.3.1 POMCP-UCD

We propose to apply the UCD framework from [22] to the action-DAGs of the POMCP belief DAG in an algorithm we call POMCP-UCD. The original UCD framework does not exploit the prior knowledge on the symmetries of the combinatorial action space. The POMCP-UCD algorithm we propose in this thesis *does* exploit this knowledge to a small degree. It does so by, during the selection phase, checking if any of the current possible sub-actions lead to parts of the action-DAG already traversed. If so, it will initialize those transpositions into the action-DAG and *then* select a sub-action using the UCD algorithm. In this way, it initializes transpositions *before* selecting an action rather than after.

Although intuitively one would like to maximally exploit the available prior and sampled knowledge, it might be that doing so can lead to sub-optimal behaviour. The authors of [22] show this to be the case in a particular example scenario. It will be interesting to study this idea further when applied to the action-DAGs of our combinatorial action space. In this thesis, we will compare this method with the POMCP-UA method that exploits the prior and sampled knowledge to a higher degree.

Sampled knowledge

The UCD framework exploits transpositional information by sharing the information obtained from the rollouts with different parts of the search tree through

transpositions. The transpositions/rollouts are found/performed through sampling of the environment (the simulations). In a way, this can be viewed as exploiting the *sampled knowledge* obtained in the simulations.

One important thing to note about the adapted score in the UCD framework is that it will never maximally use the sampled knowledge available to it. This is because the adapted score μ_d uses the (un-adapted) count n to recursively weigh its children's scores. This means the score of a transposition is weighed by the number of times this transposition was traversed (n), rather than the number of rollouts responsible for that average score. Therefore, the adapted score potentially undervalues the values of its children if the information from that value was obtained through transpositions.

Prior knowledge

Another thing to note about the POMCP-UCD algorithm is that it doesn't maximally exploit the prior knowledge either. One reason for this is that the UCD algorithm was designed with zero prior knowledge in mind. So we decided to exploit the prior knowledge only to a small degree to maintain the POMCP-UCD method in a similar spirit as the original UCD framework.

5.3.2 POMCP-UA

We propose to apply the update-all strategy from section 3.2.2 to the action-DAGs of the POMCP belief DAG in an algorithm we call POMCP with Update-All (POMCP-UA). The update-all algorithm is designed to maximally use the sampled knowledge obtained from the rollouts throughout the search tree.

Furthermore, the POMCP-UA method we propose exploits also the prior knowledge by initializing in the action-DAG every possible ascent path for any traversed leaf node. In other words, once it reaches an action-DAG leaf node, the POMCP-UA algorithm will initialize every possible transposition that can lead to that particular node.

Intuitively, we expect the POMCP-UA method to perform the best out of all the methods in this thesis. This is because this method exploits the combinatorial structure of our action space to higher degrees than any of the other methods proposed. Intuitively, exploiting more information could lead to better informed and therefore more optimal decisions.

The reason the authors of [22] don't opt for this strategy is because they show a counter-example scenario in which this strategy results in sub-optimal behaviour. We theorize that this counter-example cannot occur in the action-DAGs for combinatorial action spaces. It will therefore be interesting to see how this algorithm performs in our problem domain compared to POMCP-UCD. This

could give an indication of whether exploiting structure to a higher degree will indeed lead to better solutions in problem domains with combinatorial action spaces.

Part IV

Empirical Evaluation

Chapter 6

Methodology

This chapter contains the implementation details of the methods and simulator used for the empirical evaluation in this thesis. The solution methods and their pseudo-code are detailed in section 6.1. The implementation details of the AML investigation simulator are provided in section 6.2.

6.1 Solution Method Specification

This section contains all the implementation detail on the solution methods used in this thesis. It contains pseudo-code for all the methods and notes on the implementations. We start by describing the baseline policy to which the other methods will be compared.

6.1.1 Baseline

The baseline policy in this thesis will be a policy that does not take into account the sequential decision making aspect of the problem. It will serve as a comparison to which the POMCP-based methods can compare. On top of that, the POMCP-based methods are online methods which can be used as a way to improve upon a baseline policy (the rollout policy). So, the baseline policy of this thesis also serves as the rollout policy for the POMCP-based methods in this thesis.

The most straightforward baseline policy to use would be the *greedy* policy that greedily selects nodes in descending order of risk score. This policy is also the most likely one to use if one doesn't take into account the sequential decision making aspect. This is because the greedy policy will likely investigate the most suspicious nodes first (if the risk score is indeed a metric for suspicion).

Note that there can be multiple equivalent greedy policies if there are duplicate risk scores. So, generally speaking, our baseline will a class of greedy policies that consists of all the policies that choose nodes in descending order of

risk score.

6.1.2 POMCP

The regular POMCP method is the method as explained in section 2.3. All the other methods in this thesis are based on this POMCP method and share the procedures in algorithm 2.

Algorithm 2 POMCP Procedures

```
procedure SEARCH(h)
    repeat
        if h = empty then
            s ~ b_i
        else
            s ~ B(h)
        end if
        SIMULATE(s,h,0)
    until TIMEOUT
    return FINDOPTIMALACTION(h)          ▷ argmax over (sub-)actions
end procedure

procedure ROLLOUT(s, h, depth)
    if then γ^depth < ε
        return 0
    end if
    a ~ π_rollout(h)
    (s', o, r) ~ G(s, a)
    return r + γ ROLLOUT(s', hao, depth + 1)
end procedure
```

The regular POMCP method as defined in [17] consists of the procedures in algorithm 2 combined with the procedure in algorithm 3. This implementation has issues with large action spaces as it cannot even solve small combinatorial problems without running out of memory.

Algorithm 3 POMCP Regular

1: **procedure** SIMULATE($s, h, depth$)
2: **if** $\gamma^{depth} < \epsilon$ **then**
3: **return** 0
4: **end if**
5: **if** $h \notin T$ **then**
6: **for all** $a \in A$ **do**
7: $T(ha) \leftarrow (n_{init}(ha), \mu_{init}(ha), \emptyset)$
8: **end for**
9: **return** ROLLOUT($s, h, depth$)
10: **end if**
11: $a \leftarrow \text{argmax}_b \, \mu(hb) + c\sqrt{\frac{\log n(h)}{n(hb)}}$
12: $(s', o, r) \sim \mathcal{G}(s, a)$
13: $R \leftarrow r + \gamma$ SIMULATE($s', hao, depth + 1$)
14: $B(h) \leftarrow B(h) \cup \{s\}$
15: $n(h) \leftarrow n(h) + 1$
16: $n(ha) \leftarrow n(ha) + 1$
17: $\mu(ha) \leftarrow \mu(ha) + \frac{R - \mu(ha)}{N(ha)}$
18: **return** R
19: **end procedure**

In this thesis, we propose a memory-efficient implementation of the POMCP algorithm in which action nodes are only created once they are actually traversed during the selection phase. The pseudo-code for this memory-efficient implementation can be found in algorithm 4. The difference between these is that in the regular POMCP algorithm (algorithm 3) all the action nodes are initialized (line 7). In the memory-efficient implementation (algorithm 4) however, a node is not initialized until it is traversed (line 10). This does mean of course that the current accessible action nodes have to be temporarily initialized (line 21) in order to perform the UCT `argmax` (line 25). However, after the `argmax` the nodes can be cleared (line 26) since they don't store any new information (just the initial values).

Algorithm 4 POMCP memory-efficient

procedure SIMULATE($s, h, depth$)
  if $\gamma^{depth} < \epsilon$ **then**
    return 0
  end if
  if $h \notin T$ **then**
    return ROLLOUT($s, h, depth$)
  end if
  $a \leftarrow$ TREEPOLICY(h)
  if $ha \notin T$ **then**
    $T(ha) \leftarrow (n_{init}(ha), \mu_{init}(ha), \emptyset)$ ▷ only create nodes traversed
  end if
  $(s', o, r) \sim \mathcal{G}(s, a)$
  $R \leftarrow r + \gamma$ SIMULATE($s', hao, depth + 1$)
  $B(h) \leftarrow B(h) \cup \{s\}$
  $n(h) \leftarrow n(h) + 1$
  $n(ha) \leftarrow n(ha) + 1$
  $\mu(ha) \leftarrow \mu(ha) + \frac{R - \mu(ha)}{N(ha)}$
  return R
end procedure

procedure TREEPOLICY(h)
  for all $ha \notin T$ **do** ▷ create uninitialized action nodes
    $A_{init} \leftarrow a$
    $T(ha) \leftarrow (n_{init}(ha), \mu_{init}(ha), \emptyset)$
  end for
  $action \leftarrow \text{argmax}_b\, \mu(hb) + c\sqrt{\frac{\log n(h)}{n(hb)}}$
  for all $a \in A_{init}$ **do** ▷ free uninitialized action nodes
    $T(ha) \leftarrow$ `None`
  end for
  return *action*
end procedure

It is possible to view the memory-efficient POMCP algorithm as the regular POMCP algorithm with sparse belief tree. In this sense, the two implementations do not differ in their outcome, only in their memory footprint.

6.1.3 POMCP-AC

In this thesis, we also propose the POMCP with Action Chains (POMCP-AC) algorithm. This algorithm is an extension of the POMCP algorithm with action chains between each belief node. The algorithm shares the POMCP procedures

in algorithm 2. The pseudo-code can be found in algorithm 5.

The differences between the POMCP-AC pseudo-code and the POMCP one in algorithm 4 is that the former has to traverse the action chain in the back-propagation, `FindOptimalAction()` and `TreePolicy()` procedures.

Algorithm 5 POMCP-AC

procedure FindOptimalAction(h)
  $action \leftarrow \emptyset$
  $n \leftarrow h$
  while n is not an action leaf node **do** ▷ traverse action chain
    $a \leftarrow \text{argmax}_b V(hb)$
    $action \leftarrow action \cup a$
    $n \leftarrow na$
  end while
  return *action*
end procedure

procedure Simulate($s, h, depth$)
  if $\gamma^{depth} < \epsilon$ **then**
    return 0
  end if
  if h is a new node **then**
    return Rollout($s, h, depth$)
  end if
  $a, n \leftarrow$ TreePolicy(h)
  $(s', o, r) \sim \mathcal{G}(s, a)$
  $R \leftarrow r + \gamma$ Simulate($s', no, depth + 1$)
  $B(h) \leftarrow B(h) \cup \{s\}$
  $N(h) \leftarrow N(h) + 1$
  for $n \in$ descent path **do**
    $N(n) \leftarrow N(n) + 1$
    $V(n) \leftarrow V(n) + \frac{R - V(n)}{N(n)}$
  end for
  return R
end procedure

procedure TreePolicy(h)
  $action \leftarrow \emptyset$
  $n \leftarrow h$
  while n is not an action leaf node **do** ▷ traverse action chain
    $a \leftarrow \text{argmax}_b\, u_{UCT}(nb)$ ▷ include uninitialized actions
    if na not in tree **then**
      $na \leftarrow (N_{init}(na), V_{init}(na))$
    end if
    $action \leftarrow action \cup a$
    $n \leftarrow na$
  end while
  return $action, n$
end procedure

6.1.4 POMCP-UCD

The original UCD framework has three depth parameters that dictate how far down the search DAG the adapted variables will 'look' for relevant transpositions. The first parameter is the depth parameter d_1 for the adapted score μ_{d_1}. In our action-DAGs, the intermediary nodes do not represent actual actions that can be taken, only parts of actions. This means those nodes do not have any reward associated with them and therefore also no mean value μ. Due to this, the only sensible parametrization for μ_{d_1} is $d_1 = K$, where K is the action budget. The other two depth parameters cannot be fixed in this way. However, for the sake of simplicity and because both parameters govern the exploration factor of UCD, we will merge them into a single parameter: $d_2 = d_3 = d$.

The POMCP-UCD algorithm shares the POMCP procedures in algorithm 2 and the `FindOptimalAction()` procedure in algorithm 5. The pseudo-code for POMCP-UCD can be found in algorithm 6. The `TreePolicy()` procedure differs from the POMCP-AC one in that it uses the UCD policy rather than the UCT one. Furthermore, after traversing a node it has to check if a transposition of this node already exists in the tree. Moreover, the adapted factors off all the ancestors of newly created nodes have to be properly updated according to the equations of the adapted factors in section 3.2.1.

Also, the `Simulate()` procedure differs from the POMCP-AC one in that the adapted factors of all the ancestors of the playout node need to be updated according to the procedure described in [22].

Furthermore, this algorithm exploits the prior knowledge of the combinatorial symmetries by adding transpositions before selecting an action rather than after.

Algorithm 6 POMCP-UCD

```
1: procedure TreePolicy(h)
2:     action ← ∅
3:     n ← h
4:     while n is not an action leaf node do
5:         for all a ∈ A not in current descent path do      ▷ add transpositions
6:             if there is a transposition n′ in the tree then
7:                 na ← n′
8:             end if
9:         end for
10:        a ← argmax_b u_UCD(nb)                          ▷ Include uninitialized actions
11:        if na not in tree then
12:            if there is a transposition n′ in the tree then
13:                na ← n′
14:            else
15:                na ← (N_init(na), V_init(na))
16:            end if
17:            UpdateAdaptedFactorsAncestorsAfterNewNode(na)
18:        end if
19:        action ← action ∪ a
20:        n ← na
21:    end while
22:    return action, n
23: end procedure
```

Original UCD framework

In this thesis, we will compare the POMCP-UCD algorithm defined above with a version of this algorithm that follows the original UCD framework more closely. The original UCD framework does not exploit the prior knowledge of the symmetries and so the POMCP-UCD Original method defined in algorithm 7 adds transpositions only *after* encountering them.

Implementation

Another important thing to note about the UCD framework is that the backpropagation procedure as described in [22] is incomplete in certain scenarios. In this paper, the authors show the UCD framework can be more efficiently implemented if the adapted values are stored in the nodes and updated during the backpropagation phase (rather than using the recursive definitions). They state the adapted score μ_d can be updated by looking at the variation $\Delta\mu_d(e)$ of $\mu_d(e)$ induced by the playout ($\Delta\mu_d(e) = 0$ if e is not an ancestor of the leaf node from

Algorithm 7 POMCP-UCD Original

procedure Simulate($s, h, depth$)
  if $\gamma^{depth} < \epsilon$ **then**
    return 0
  end if
  if h is a new node **then**
    return Rollout($s, h, depth$)
  end if
  $action, a \leftarrow$ TreePolicy(h)
  $(s', o, r) \sim \mathcal{G}(s, action)$
  $R \leftarrow r + \gamma$ Simulate($s', hao, depth + 1$)
  $B(h) \leftarrow B(h) \cup \{s\}$
  $N(h) \leftarrow N(h) + 1$
  for $n \in$ descent path **do**
    $N(n) \leftarrow N(n) + 1$
  end for
  UpdateAdaptedFactorsAncestorsAfterRollout(hao)
  return R
end procedure

procedure TreePolicy(h)
  $action \leftarrow \emptyset$
  $n \leftarrow h$
  while n is not an action leaf node **do**
    $a \leftarrow \text{argmax}_b\, u_{UCD}(nb)$ ▷ Include uninitialized actions
    if na not in tree **then**
      if there is a transposition n' in the tree **then**
        $na \leftarrow n'$
      else
        $na \leftarrow (N_{init}(na), V_{init}(na))$
      end if
      UpdateAdaptedFactorsAncestorsAfterNewNode(na)
    end if
    $action \leftarrow action \cup a$
    $n \leftarrow na$
  end while
  return $action, n$
end procedure

which playout was performed). They use the following formula for updating the adapted score:

$$\Delta\mu(e) = \frac{\sum_{f \in c(e)} \Delta\mu(f) \times n(f)}{n'(e) + \sum_{f \in c(e)} n(f)} \tag{6.1}$$

where $c(e)$ are the children of e and $n'(e)$ is the initial count for e.

However, we noticed in some scenarios this formula is incomplete. Let's derive the equation:

$$\mu_t(e) = \frac{\sum_{f \in c(e)} \mu_t(f) \times n_t(f)}{n'(e) + \sum_{f \in c(e)} n_t(f)} \tag{6.2}$$

$$\mu_t(e) = \frac{\sum_{f \in c(e)} \Big(\mu_{t-1}(f) + \Delta\mu(f)\Big) \times n_t(f)}{n'(e) + \sum_{f \in c(e)} n_t(f)} \tag{6.3}$$

$$\mu_t(e) = \frac{\sum_{f \in c(e)} \mu_{t-1}(f) \times n_t(f)}{n'(e) + \sum_{f \in c(e)} n_t(f)} + \frac{\sum_{f \in c(e)} \Delta\mu(f) \times n_t(f)}{n'(e) + \sum_{f \in c(e)} n_t(f)} \tag{6.4}$$

$$\mu_t(e) \neq \mu_{t-1}(e) + \frac{\sum_{f \in c(e)} \Delta\mu(f) \times n_t(f)}{n'(e) + \sum_{f \in c(e)} n_t(f)} \tag{6.5}$$

The reason for the inequality in the last line is because $n_t(f)$ does not necessarily equal $n_{t+1}(f)$. Those two are only equal if f is not part of the descent path. So, the original formula stated in the paper [22] is incomplete if e has children that are part of the descent path.

If the node e has a child i that is part of the descent path, we will use the following formula:

$$\mu_t(e) = \frac{\mu_{t-1}(e) \times \Big(n'(e) + \sum_{f \in c(e)} n_t(f) - 1\Big) + \mu_{t-1}(i)}{n'(e) + \sum_{f \in c(e)} n_t(f)} + \frac{\sum_{f \in c(e)} \Delta\mu(f) \times n_t(f)}{n'(e) + \sum_{f \in c(e)} n_t(f)} \tag{6.6}$$

which is still defined in terms of values stored at the nodes (so no need for the recursive definitions). The difference between the two update formulas is small but stack up with repeated iterations. After a small amount of time the differences can be significant.

Furthermore, they state that to update n_d you have to add one to every edge on the descent path and every edge in $a_d(l)$. Here $a_d(l)$ is the set of ancestors of node l at a distance at most d from l and l is the leaf node from which playout is performed. This would be complete if the leaf node l were the only one for which the count n is incremented. However, the count n is incremented for every edge in the descent path. So, the adapted count n_d should also be incremented for every edge in $a_d(e)$ where e is every edge in the descent path.

6.1.5 POMCP-UA

In this thesis, we also propose the POMCP-UA method which consists of the update-all strategy applied to the action-DAGs. This algorithm shares the POMCP procedures in algorithm 2 and the `FindOptimalAction()` procedure in algorithm 5. The pseudo-code for POMCP-UA can be found in algorithm 8). This method is similar to POMCP-UCD but it initializes all possible transpositions after encountering a leaf node and uses the update-all strategy for backpropagation (taking care not to update a node more than once).

Algorithm 8 POMCP-UA

procedure SIMULATE($s, h, depth$)
  if $\gamma^{depth} < \epsilon$ **then**
    return 0
  end if
  if h is a new node **then**
    return ROLLOUT($s, h, depth$)
  end if
  $action, a \leftarrow$ TREEPOLICY(h)
  INITIALIZEEVERYPOSSIBLEPERMUTATIONOF($action$)
  $(s', o, r) \sim \mathcal{G}(s, action)$
  $R \leftarrow r + \gamma$ SIMULATE($s', hao, depth + 1$)
  $B(h) \leftarrow B(h) \cup \{s\}$
  $N(h) \leftarrow N(h) + 1$
  UPDATEALLANCESTORSAFTERPLAYOUT(hao)
  return R
end procedure

```
procedure TreePolicy(h)
    action ← ∅
    n ← h
    while n is not an action leaf node do
        a ← argmax_b u_UCT(nb)                    ▷ Include uninitialized actions
        if na not in tree then
            if there is a transposition n′ in the tree then
                na ← n′
            else
                na ← (N_init(na), V_init(na))
            end if
            UpdateAllAncestorsAfterNewNode(n)
        end if
        action ← action ∪ a
        n ← na
    end while
    return action, n
end procedure
```

6.2 Simulation Specification

This section will detail the specifics of the generative model used in this thesis. The generative model has the form:

$$(s_{t+1}, o_{t+1}, r_{t+1}) \sim \mathcal{G}(s_t, a_t) \tag{6.7}$$

where the state $s_t = \{G, l^r, l^o_t, f^r, f^o_t, f^c_t\}$ has the form as described in chapter 4. It is worth noting that in the procedures outlined below the risk score of the nodes is sometimes used. We assume the risk score will be a function of the observed feature values f^o_t and can therefore be derived from the state s_t. The action a_t consists of a set of nodes with size equal to the action budget K. The nodes in a_t are the ones under formal investigation and are the starting points of the investigation procedure.

6.2.1 Investigation

We will describe the investigation procedure for a single node v under formal investigation. If there are more then one node under investigation (budget $K > 1$), the procedure works in exactly the same way as the single node investigation just applied to all the nodes $v \in a_t$.

Determining neighbours to investigate

The first step of the investigation procedure is determining what neighbours of v are to be investigated as well. Neighbours of a node v are only investigated if node v has a true illicit label. This is supposed to simulate that investigation into v triggered investigations into its neighbourhood, which we don't necessarily expect to happen if a node is not fraudulent. So, if a node v has a true licit label, only an investigation into v occurs. Otherwise, the following procedure will determine what neighbours to investigate.

We will perform a search through all the d-hop neighbours of node v where $d \leq d_I$ (d_I being the investigation depth parameter of our model). Out of this d_I-hop neighbourhood, the $K_I - 1$ nodes with the highest risk score are selected for investigation (K_I being the investigation budget parameter of our model). The only exception to the above rule is that nodes that already have been fully investigated (f_t^c is 1 for all features) are ignored.

Let's denote the set of selected nodes for investigation $V_I(v)$. This set includes the original node v, which means the size of this set is $|V_I(v)| \leq K_I$, with the lesser than only occurring if the d_I-hop neighbourhood of v is smaller than K_I. Combining the sets $V_I(v)$ for all $v \in a_t$ is done in a set like union:

$$V_I = \{V_I(v) \cup V_I(u) | u, v \in a_t\} \tag{6.8}$$

This means there are no duplicate nodes in the set V_I, meaning nodes are never investigated twice in the same time step.

Note also that this procedure does not account for the fact that a node u can be in the set $V_I(v)$ if both $u, v \in a_t$. This feature was retained as a way to simulate the double work that potentially can happen when investigations overlap.

Performing investigations

Once the set V_I of nodes to be investigated is determined, the actual nodes $w \in V_I$ are investigated. The actual investigation procedure consists of a series of n independent Bernoulli trials (where n is the number of features per node). A 'success' outcome denotes the case in which the true value is observed. In this case we set $f_t^o(x) = f^r(x)$ and $f_t^c(x) = \textit{correct}$ for the particular feature x of node w. In the case of 'failure', we do not change anything.

The probability of a 'success' p is dependent on the type of node w investigated. If w is one of the nodes under formal investigation $w \in a_t$ the probability of 'success' could differ from nodes $w \notin a_t$. However, in the experiments performed in this thesis, both probabilities were set to 1 (investigations always reveal all the possible information on a node).

6.2.2 Labeling

Labeling nodes as illicit simulates the process of filing a Suspicious Activity Report (SAR) to the public prosecution service. In real life, this typically will only happen for the nodes $v \in a_t$ under formal investigation. However, in our model, we will label all nodes under investigation $v \in V_I$. This is not necessarily a realistic assumption, since in real life typically only the node that is formally under investigation can receive a SAR.

In real life suspicious activity from a node v might be encountered during an investigation into node $a^i \in a_t$. In this case, a new formal investigation would probably be launched into node v. In real life, however, investigations don't take a fixed amount of time. The investigation into node v will probably proceed much more swiftly than it would normally because part of the investigation was already performed during the investigation into a^i. In the fixed time-step investigations we use, we will somewhat model this by including the SAR of the suspicious node v in the investigation into a^i.

The nodes $v \in V_I$ are labeled with a sequence of independent Bernoulli trials where 'success' indicates that the node is labeled as illicit. A 'success' outcome occurs with a probability p that is determined through a number of different dependencies.

Determining label probability

For simplicity, we will assume our investigators don't falsely label nodes as illicit. In other words, if a node's v true label is licit, we assume the probability of labeling it illicit $p(v)$ is 0. If the true label is illicit, the probability is determined in the following manner.

The probability $p(v)$ starts with a base labeling probability of $p(v) = 0.3$. Based on the information known about this node $f_t^c(v)$ the probability is increased in the following way:

$$p(v) = p(v) + 0.3 * \frac{|\{f_t^c \in f_t^c(v) | f_t^c = 1\}|}{|f_t^c(v)|} \tag{6.9}$$

that is, the probability is increased by 0.3 times the fraction of (assumed to be) correct features.

Next, we want to increase the probability of a lot of information is known about the neighbours $N(v)$ of v. Per (1-hop) neighbour $u \in N(v)$ we increase the probability with a certain amount based on the information known according to:

$$p(v) = p(v) + \sum_{u \in N(v)} \frac{0.2}{|N(v)|} * \frac{|\{f_t^c \in f_t^c(u) | f_t^c = 1\}|}{|f_t^c(u)|} \tag{6.10}$$

Lastly, if the node v has neighbours with a true illicit label, we want the probability to increase if those neighbours are already found to be illicit (finding a fraudulent node is easier if its fraudulent network is already discovered). We do this by increasing the probability with 0.2 times the fraction of neighbours found to be illicit over the amount of true illicit neighbours:

$$p(v) = p(v) + 0.2 * \frac{|\{w \in N(v) | l_t^o(w) = illicit\}|}{|\{w \in N(v) | l^r(w) = illicit\}|} \tag{6.11}$$

All in all, if the node v has illicit neighbours who have all been found already, the probability of labeling v as illicit $p(v)$ can be a maximum of 1. If the node has no illicit neighbours, the probability can be a maximum of 0.8. This simulates that illicit nodes that are not part of a fraudulent network are more difficult to find.

6.2.3 Sampling Scenarios

During our experiment, we may wish to fix the stochasticity in the environment (model). The main stochasticity in our environment manifests itself in the labeling of nodes as fraudulent or not. A node v is labeled fraudulent or not with a probability $p(v)$ described above. This is implemented by sampling a uniform random number ϕ from the range $[0, 1]$ and comparing it with $p(v)$ (if $\phi \leq p(v)$ the node is labeled fraudulent). This is an implementation of a Bernoulli trial.

We can fix the stochasticity of the environment by sampling a *scenario* Φ. A scenario Φ is obtained by sampling and initial state s_0 and uniform random numbers ϕ_i^v for every node v before any simulation occurs. In other words, for a node v we sample the numbers $\Phi^v = \{\phi_1^v, \phi_2^v, ...\}$ which determine the outcomes of the stochastic labeling of this node (where $\Phi = \{\Phi^v | v \in V \text{where } l^r(v) == illicit\}$. So, a scenario determines the outcome of the first investigation into node v (with ϕ_1^v) and when the first investigation resulted in erroneous labeling, it determines the outcome of the second investigation (with ϕ_2^v), and so on.

This is implemented by fixing the seed of the Python random number generator before sampling a sequence of uniform random numbers $\Phi^v = \{\phi_1^v, \phi_2^v, ...\}$ for every true illicit node in the network. The sequence Φ^v is terminated when a draw ϕ_i^v is smaller than the base labeling probability since in this case the node v will always be labeled illicit, after which no more draws are necessary.

Note that all other stochasticity in the environment (observing features) draws from the same random number generator whose seed was fixed in the procedure above.

6.2.4 Initialization

Initialization of the state and environment were already discussed in section 4.2.3. We will briefly go over the specific parameters used in the initialization of f_0^o and f_0^c.

For the observed feature values f_0^o, we take the real values f^r and add noise to them. The noise we add is drawn from a random normal distribution with mean 0 and standard deviation 1. This noise is not added to every single observed feature value. Instead, we perform a sequence of independent Bernoulli trials per feature per node that determine what values to add noise to. The 'success' outcome occurs with probability 0.5 and denotes that this feature is indeed correct and is not noisy. The 'failure' outcome denotes an error occurred in the acquisition method of this feature and noise is added to this particular feature for this particular node.

The features we assume are correct are also determined through a sequence of independent Bernoulli trials (as this mirrors our prior knowledge of the observing process above). In this thesis, we assume we know the probability with which we add noise (0.5 in this case) and therefore the Bernoulli trials for f_0^c also occur with probability 0.5. The 'success' outcome means we believe the feature to be correct.

The initialization of the transaction network is viewed as part of the problem instance rather than the environment in our thesis. The specifics of its initialization are therefore described in chapter 7.

6.2.5 Lookahead Model

In the above procedures, we sometimes rely on the true illicit label of a node v. In real life, this knowledge is not known (this would defeat the purpose of our investigations). Therefore, in the generative model used for the lookahead search of the POMCP-based methods, the above model needs to be adjusted slightly.

For the lookahead model $\mathcal{G}_{lookahead}(s_t, a_t)$, the true labels l^r are derived from the risk scores (which are assumed to be based on f_t^o). For our experiments, this is done by applying a thresholding function to the risk score with a threshold of 0.5 (if the risk is above 0.5, assume the nodes true label is illicit). This means in the lookahead model the simulation of the investigations depends on the knowledge we have so far.

In conclusion, there are really two types of models used in this thesis: the regular model $\mathcal{G}(s_t, a_t)$ and the lookahead one $\mathcal{G}_{lookahead}(s_t, a_t)$. The regular model is used as the environment in which we evaluate the performances of our methods. Ideally, this would be performed in real life, but in our case this would not be feasible. So, the regular model simulates evaluating our methods in real

life and can therefore depend on things like the true labels.

The lookahead model rather simulates how we would model our environment with the information we would have in real life. This can therefore not depend on the true labels, as this is not information we would have in a real-life planning scenario.

Chapter 7

Setup

This chapter will contain all the information on the experimental setup. Section 7.1 will go through the problem instance on which the methods will be evaluated. This is followed by a description and motivation of the experiments that will be performed in this thesis in section 7.2. Finally, section 7.3 will discuss the methods of evaluation with which we will compare our solution approaches.

7.1 Problem Instance

The environment we are using is a highly stochastic environment where the differences between optimal and sub-optimal policies can be easily overshadowed by unlucky simulation runs. This means, in order to properly evaluate the performances of the different algorithms, we have to take averages over many different runs. In order to do this in a reasonable amount of time, we will use a relatively small transactional graph.

7.1.1 Transactional Graph

The problem scenario in question consists of finding fraud in a transactional network with 57 nodes and 68 edges. The transactional network is obtained from the AMLSim (described in 4.2.3) sample dataset with 1000 accounts and 100.000 transactions. At first, a subnetwork was sampled from this 1000 node network by inducing a subgraph of neighbours centered around a particular node n within a given radius. Due to the high connectivity of the AMLSim graph, we found we required very small values of the radius to induce a small enough graph. However, a small diameter of the graph does not lead to interesting behaviour of our investigation simulator. Therefore, we decided to remove high degree nodes from the graph and sample again around a particular node. In this way, we ended up with a 57 node graph by removing the top 82% high degree nodes and sampling in a radius of 5 around node number 9.

Labels

As far as node labels go, the AMLSim dataset comes with illicit/licit labels. These labels are obtained from several AML typologies that are purposefully inserted into the graph. However, when subsampling a 57 node network from the 18% lowest degree nodes from a 1000 node network, we found those typologies would not stay intact. Therefore, the labels provided by the AMLSim dataset were effectively random and did not lead to a very interesting search landscape. This is because our investigation model was designed with the minimal assumption that fraudulent behaviour would occur in clusters. Picking 7 illicit nodes at random from a 57 node network would most often not result in clusters of more than 2 nodes. According to our assumption in modelling the investigations, this would not lead to a very realistic problem scenario.

Moreover, if most illicit nodes weren't in clusters, most investigations would behave independently. This would result in most actions involving illicit nodes to perform equally. Suppose we have a search space where many different actions are all equally optimal. All the POMCP-based methods will now have no problem finding one of the optimal actions and will therefore perform exactly the same. This would not lead to an interesting comparison of the methods proposed in this thesis.

For those reasons, we decided to adapt the AMLSim labels by initializing 7 illicit nodes, that are divided into two groups of three and a single group of one (see figure 7.1). This can be thought of as two occurrences of fan-in/fan-out AML typologies of three nodes each, together with a single isolated node. The AML typologies keep a sense of realism in the illicit nodes we picked and the 3-3-1 partition of the nodes would allow us to create a solution landscape that is sufficiently complex to properly compare our methods.

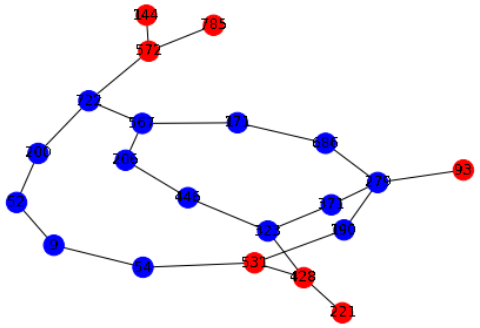


Figure 7.1: The structure of illicit nodes in our scenario. Only the nodes and edges that are on the shortest paths between any two illicit nodes are drawn here. Illicit nodes are in red, whereas licit nodes are drawn blue.

# of nodes	57
# of edges	68
# of illicit nodes	7
Action Budget K	3
Investigation Budget K_I	3
Investigation Depth d_I	1
Discount Factor γ	0.5

Table 7.1: Summary of the scenario parameters.

7.1.2 Problem Parameters

What the optimal policy is does not only depend on the topology of the illicit nodes in the network. It also depends on the problem parameters, namely the investigation simulator parameters (introduced in chapter 4) and the action budget size K.

The action budget (how many nodes can be investigated per time step) for this scenario is set to $K = 3$. If the action budget were $K = 2$, we wouldn't be able to distinguish between the POMCP-UCD and POMCP-UA methods (since these behave the same for action chains that are only of length 2). The same goes trivially for an action budget of $K = 1$. So, we choose the next smallest action budget (to limit computational resources) that allows us to properly compare all our algorithms, which is $K = 3$

As for the other investigation simulator parameters, the investigation budget is set to $K_I = 3$. This means for every selected node two of its neighbours are investigated as well. The investigation depth is set to $d_I = 1$, which means only first order neighbours are considered. By setting these parameters as such, there is a single optimal first action to take in this problem instance. This action is the one where both the middle nodes in the two clusters of illicit nodes are selected (nodes 572 and 428 in figure 7.1), together with the isolated node 93. This way it is possible to find all 7 illicit nodes in the first time-step. A summary of the scenario's characteristics can be found in table 7.1.

Finally, since the real life time between time-steps can be anything from days to months, we opted for a discount factor of $\gamma = 0.5$. This makes sure there is extra emphasis in the POMCP-based methods on finding good actions as fast as possible. Intuitively, this discount factor means finding fraud in the next time step, rather than in the current one, is valued half as much.

7.1.3 Risk Score

The greedy policy is defined as the policy that investigates nodes in descending order of their risk score. This risk score will typically be obtained from some

classification algorithm on the nodes in the transaction network. This will usually be performed on the bases of some set of features and/or the graph structure around these nodes. In reality, those features could be incomplete or incorrect, resulting in erroneous risk scores. Part of a real life investigation could be to observe new features or correct erroneous ones, which as a result might update the risk score.

In our case, transactional graph is obtained from the AMLSim example data set. This data set does not have extensive features that can be used to build a realistic classifier for licit/illicit behaviour. Instead, we will greatly simplify this part of the problem by assigning a risk score of 1 (the highest value) to illicit nodes and a score of 0 (the lowest value) to licit ones. We do this based on the true labels of the transactional network. This will result in a 'perfect' risk score that exactly describes the fraudulent nodes in the network. Note that this also results in a risk score that is static (doesn't change based on the investigations).

This is not necessarily a very realistic scenario. However, the 'perfect' risk score does allow us to evaluate whether decision making algorithms can increase performance *even if* the classification of fraudulent nodes is 100% accurate. Our intuition is that if the risk score is erroneous, sequential decision making can potentially allow for even larger improvements. This was partly discussed in our motivation in section 1.3.2, where we reasoned that if the incorrect risk scores of two nodes were caused by the same underlying issue, it might be unfavorable to investigate them at the same time.

Note on the baseline policy

The baseline in this thesis is the class of greedy policies. These policies choose to investigate nodes in 'greedy' order of descending risk score. However, in our problem instance the risk scores are static. This means that the greedy policies in this problem instance are also static.

Moreover, in our problem instance the risk scores for all illicit nodes is 1 (and 0 for licit ones). This means that a greedy policy would be any policy that investigates the illicit nodes before the licit ones. Our problem instance does not differentiate between illicit nodes however and therefore any ordering among those can be considered greedy. This means in our problem instance the greedy baseline turns into a class of possible greedy baseline policies. For 7 illicit nodes there are exactly $7! = 5040$ possible greedy policies (if we assume the order of the licit nodes following the 7 illicit ones doesn't matter).

7.2 Experimental Setup

The goal of our experiments will be to try and answer some of the research questions posed in chapter 1. In order to do so, we will structure our experiments in the following three categories:

- **Comparing POMCP Implementations** This contains one experiment which is designed to show the difference in memory footprint between the regular and the memory efficient implementation of the POMCP algorithm.
- **Exploiting Combinatorial Structure** This category contains experiments that are designed to compare the performance of methods that exploit the combinatorial structure of the action space in their architecture and/or algorithms.
- **Baseline Comparison** Here we will take the best performing approach in the previous category and scale up the computational resources for it. We do this to show it can outperform even the highest performing baseline.

In particular for the second category, we will be comparing a range of different methods as discussed in chapter 5. We will now go over the different experimental categories in more detail.

7.2.1 Comparing POMCP Implementations

This category contains a single experiment that compares the performance of the regular and the memory efficient implementation of the POMCP algorithm. The original POMCP implementation as described in [17] scales very badly with increasing action space size. For this reason we proposes a more memory efficient implementation of the POMCP algorithm that only initializes action nodes when it traverses them in the search tree. We will run an experiment designed to show the difference in memory footprint between the two implementations.

7.2.2 Exploiting Combinatorial Structure

This category contains two experiments that are designed to investigate the benefits of exploiting the combinatorial structure of the action space. This section involves all the POMCP-based solution methods proposed in this thesis.

POMCP-UCD

The POMCP-UCD algorithm is based on the UCD algorithm described in chapter 3 and can be viewed as an application of UCD to the POMCP algorithm. In the

regular UCD algorithm, transpositions are only discovered once an action is taken and a state is reached that was encountered before. In this way it does not exploit the prior knowledge of the symmetries in the combinatorial action space.

The POMCP-UCD algorithm we propose in this thesis *does* exploit this prior knowledge to a small degree. In this experiment we will compare this POMCP-UCD algorithm to one that follows the original UCD framework strategy of not exploiting the prior knowledge. This will allow us to investigate the benefit of this adaption in our POMCP-UCD algorithm.

All Methods

This experiment is designed to compare the performance off all the methods proposed in this thesis. The methods proposed in this thesis are all algorithms designed to exploit the combinatorial structure of the action space in various degrees. We will compare all these methods and how their performance scales when we increase the amount of simulations per time step (how fast it converges to 'optimal').

The main focus of this experiment is to compare the performance of the different POMCP-based methods. However, we will also compare these methods with the average performance of the greedy baseline class of policies. This will give a sense of how the difference in performance of the POMCP-based methods compares to the baseline greedy policies.

Scenarios

The experiments in this category have two sources of stochasticity: the methods and the environment. In order to properly compare the performance of the methods we fix the stochasticity of the environment by sampling scenarios beforehand.

The methods in the experiments of this category are compared while keeping the environment scenarios fixed. This fixes the variance of our environment and allows us to focus on the variance of the algorithms. In order to avoid scenario specific differences between the methods, we compare them on four different scenarios. This hopefully allows us to identify scenario independent trends between the methods.

Research Questions

Both experiments in this category try to answer the second research question: *How can we exploit the combinatorial structure of the possible decisions to improve the performance of POMDP solution approaches?*. The first experiment evaluates

how exploiting the prior knowledge affects the POMCP-UCD algorithms performance. The second experiment evaluates algorithmic and architectural exploits of the structure with varying degree by comparing the different methods proposed in this thesis.

In the second experiment the performance of the methods are also compared to the average performance of the baseline class of policies. In this way it also tries to partially answer the third research question: *How do sequential decision making approaches compare to methods that ignore the investigation dynamics?*

This third question is explored in more depth in the last experimental scenario.

7.2.3 Baseline Comparison

This category is designed to investigate the limits of sequential decision making approaches. It consists of a single experiment in which we compare the best performing POMCP-based method in the previous experiment with the upper bounds obtained from the greedy policies. We expect that, given enough search time, the POMCP-based methods can reach the upper bounds of the greedy policies.

When the search time for a POMCP-based method is increased, the policy found will converge closer to the optimal one. Consequently, we also expect the variance of the POMCP-based method to decrease as we increase the search time significantly. Furthermore, there is a fundamental difference between the baselines in this thesis and the POMCP-based methods: the baselines are static policies. So, comparing the two on a single static environment scenario (which is deterministic) would not be an interesting comparison.

For the reasons outlined above, the structure of this experiment differs from the previous category in that it includes the variance of the environment. It does so by running the best performing POMCP-based method for a single run on 100 different scenarios and comparing its performance to the upper bound of the greedy policies. This allows us to evaluate the two methods and how they manage the stochasticity of the environment.

Note that comparing it to the upper bound of the greedy policies is not necessarily a fair comparison. This would be equivalent to choosing the optimal greedy policy for a particular scenario before running it, which is impossible. This is potentially unfair because the optimal greedy policy for a particular scenario might not be the optimal greedy policy for other scenarios. However, our aim will be to show that the POMCP-based methods will perform really well even against this slightly unfair baseline.

Research Questions

This experiment aims to answer the third research question: *How do sequential decision making approaches compare to methods that ignore the investigation dynamics?*

We can view the class of greedy policies as a policy that doesn't take into account the investigation dynamics of the AML process. We aim to show the potential of the sequential decision making approaches by showing it can reach the upper bound of the greedy policies.

7.2.4 Exploration Constant

The exploration constant is a hyper-parameter all POMCP-based methods share. This constant determines the amount of exploration vs exploitation that should occur during the selection phase of the algorithm (a high exploration constant means more exploration and vice versa). The value of this hyper-parameter has direct effect on the performance of the algorithm and therefore should be optimized over. Unfortunately, the optimal value of this constant is dependent on the relative differences of the value estimate and the exploration term. All our algorithms either use different value estimate functions (regular vs adapted score), or different exploration terms (regular vs adapted simulation count). This means we cannot assume the optimal exploration constant for one algorithm will also be optimal for another.

So, experiments described above that compare performance of algorithms will be performed over a range of exploration constants. In [17] they performed all the experiments with a constant:

$$c = R_{hi} - R_{lo} \tag{7.1}$$

where R_{hi} was the highest return achieved during sample runs with $c = 0$ and R_{lo} lowest return achieved during sample rollouts. In our case, the highest possible return is 7 (which is equivalent to finding all fraud in the first time step) and the lowest possible return is in theory 0. So in order to make sure we cover a broad range of interesting exploration constants, we will compare performances for the values: $c = \{1, 2, 3, 4, 5, 6, 7\}$.

7.3 Methods of Evaluation

In the experiments outlined above, we will evaluate the different algorithms in their sample efficiency, reward obtained and time to finish. This section will discuss these in more detail, starting with the sample efficiency.

7.3.1 Sample Efficiency

POMCP-based methods typically have a termination condition that is dependent on the search time per time step. The search time, or execution time, determines the 'real-time constraint' of the online algorithm. For POMCP-based methods, execution time is determined by the execution time for a single iteration of the tree search phases (selection, expansion, evaluation and backpropagation).

In our thesis, the POMCP-based algorithms all differ in execution times for the selection, expansion and backpropagation phases. Moreover, the simulator used is simplistic enough that a rollout in it is faster than the above mentioned tree search phases for most of our algorithms. This means those phases (selection, expansion and backpropagation) are the bottlenecks of those algorithms.

Comparing the performance of these algorithms with a fixed execution time will result in different performances due to one algorithm being faster than another. An algorithm being faster means it can perform more search tree iterations, resulting in more rollout 'samples' obtained from the environment. In this thesis, we opt to compare our methods in terms of sample efficiency rather than execution time. This is achieved by using as a termination condition the amount of search tree iterations, or *simulation runs*, rather than execution time. This is done for two reasons.

The first reason is that execution time is very heavily implementation dependent. Rather than spending a lot of time separately optimizing every single algorithm, we opt to compare them in terms of sample efficiency (which is not dependent on how time-optimized an implementation is).

The second (and main) reason we opt for sample efficiency has to do with real life applicability. We compare our algorithms on relatively small problem instances with a relatively simple simulator. It is not unlikely that on real life problem instances, with a high fidelity simulator, a single rollout in the environment could be the main bottleneck of the execution time. In this case sample efficiency and execution time are interchangeable, since the selection, expansion and backpropagation phases no longer dominate the execution time of an algorithm.

7.3.2 Episodic Return

In this thesis we evaluate our methods according to the total discounted return of an episode (total reward). This metric directly relates to how many and how fast we detect illicit nodes. Due to the small discount factor (0.5), this metric is highly dependent on how good the algorithm performs in the first steps of the episode. This is therefore a good measure to compare how good algorithms are at finding the best initial action to perform (which by construction of our scenario described in 7.1 is unique).

7.3.3 Episode Length

Due to this discount factor, the total discounted return is not a good measure of how well the algorithm performs in the last few steps of an episode. For this we use as a measure the time-step at which point all the illicit nodes (of which there are 7) have been found. We call this metric the episode length (since there is no point in continuing the episode when no more illicit nodes need to be uncovered). This metric predominantly relates to how good an algorithm is at finding the last couple of illicit nodes at the end of an episode.

Chapter 8

Results

This chapter will discuss the results of the experiments laid out in chapter 7. We will follow the same structure as in that chapter. Section 8.1 will compare implementations of the POMCP algorithm. Section 8.2 will compare the performance of methods that exploit the combinatorial structure of the action space. After that, section 8.3 will compare the best performing POMCP-based method with the baseline policies.

Section 8.1 compares different implementations of the POMCP algorithm: POMCP-regular and POMCP-memory efficient. Throughout the rest of this thesis, when we refer to the POMCP algorithm, we will refer to the memory-efficient one.

Finally, some of the experiments in section 8.2 are performed on 4 different scenarios of the environment. For the sake of brevity, we only show the results of scenario 4 in this chapter. The other scenarios can be found in appendix E. We will be discussing trends that occur across the scenarios and choose to show scenario 4 as this scenario results in figures with the least amount of clutter.

8.1 Comparing POMCP Implementations

In this category we compare the memory footprint of the regular POMCP implementation versus the memory-efficient one. This is done by running both implementations for a single time step and recording the memory used during the tree search. In figure 8.1 the results for four different budget values $K = \{1, 2, 3, 4\}$ are shown as a function of simulations per time step (search time). Shown are averages and the 95% confidence interval over 10 runs.

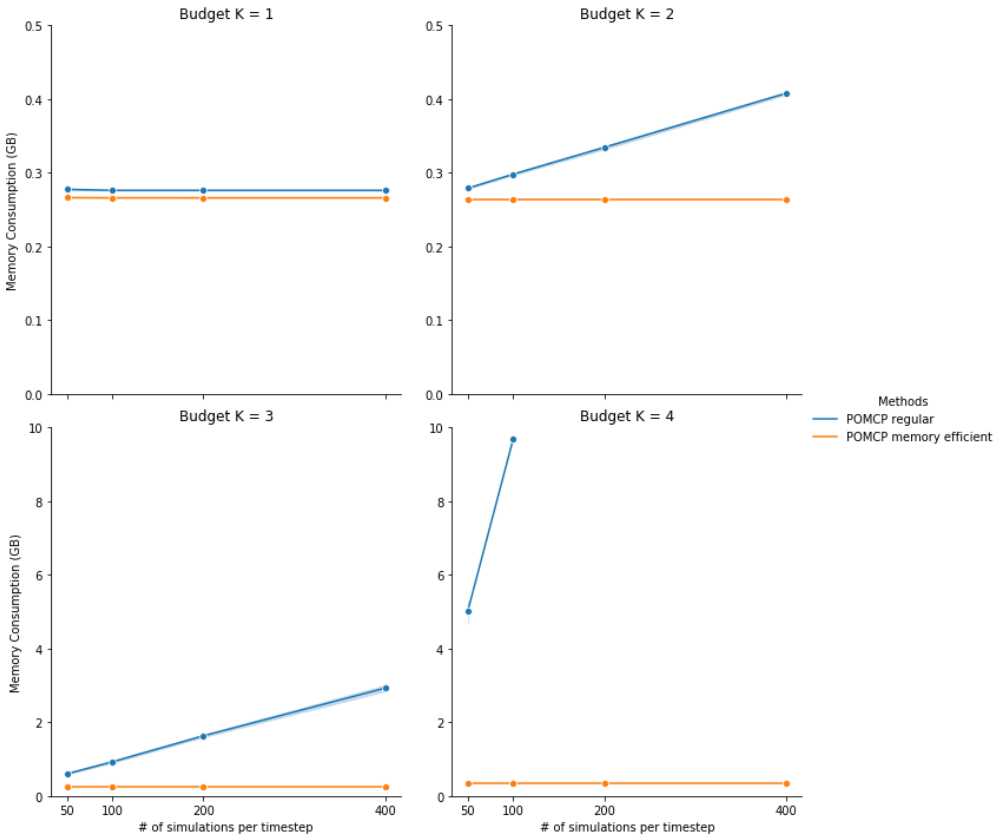


Figure 8.1: Memory consumption after the first time step for two different implementations of the POMCP algorithm. Results are obtained with $c = 2$.

The figures show an apparent linear increase in memory with increasing simulations per time step. The slope of this linear increase seems to depend on the budget size K (higher budget equals steeper slope). What is evident from the figures is that the regular POMCP implementation quickly becomes unusable for larger values of the budget K. Note that the regular POMCP exceeded 10 gigabytes of memory for $K = 4$ with only 100 simulations per time step. This is on a relatively small (compared to real-life) network of 57 nodes as well.

In contrast, the memory-efficient implementation of POMCP seemed to hold a steady memory footprint. The memory usage barely changed as a function of simulations per time step. Moreover, the memory footprint only increased slightly from somewhere around 0.26 gigabytes to 0.34 gigabytes as a function of budget K.

It is also worth noting that in this particular case, the execution times for the memory-efficient implementation was also orders of magnitude faster than the regular implementation. This was due to a quick optimization that is possible

if the initial values are the same for all nodes: $\mu_{init} = 0, n_{init} = 0$. In this case, we don't actually have to initialize new nodes during the selection of an action. Instead, we can create a single 'dummy' node with $\mu = \mu_{init}, n = n_{init}$ that represents all uninitialized nodes collectively. If the uninitialized nodes can number into the millions or more, this is not only more memory-efficient, but also more time-efficient.

So, the memory-efficient implementation performs exactly the same as the POMCP one but uses less memory and less time. This will therefore be the implementation used for the rest of the experiments. Throughout the rest of this thesis, we will be referring to the memory-efficient POMCP implementation as POMCP.

8.2 Exploiting Combinatorial Structure

The results in this section are obtained on four different scenarios of the environment. The four different scenarios are obtained by seeding the Python random number generator with the number of the scenario. For example, scenario 1 is obtained by seeding the RNG with a seed of 1. Scenario 2 is obtained with seed 2, etc. For detail on how the scenarios are drawn we refer to section 6.2.3. For brevity, the results in this section are only shown for scenario 4. The other scenarios can be found in appendix E.

In this section, we mainly compare the POMCP-based methods with each other and the average greedy performance. For this reason, we use the median greedy policy as the rollout policy for the POMCP-based methods.

8.2.1 POMCP-UCD Implementation

This section compares the POMCP-UCD method with the POMCP-UCD Original method. The Original version does not exploit the prior knowledge of the combinatorial structure of the action chains before choosing an action in the selection phase of the tree search. Figure 8.2 shows the two POMCP-UCD versions with a depth parameter of $d = 3$ (referred to as POMCP-UCD3). Shown are averages and 95% confidence intervals over 50 runs for exploration constants $c = \{1, 2, 3, 4, 5, 6, 7\}$. The results are obtained with 1100 simulations per time step. We chose to compare the two versions with a depth parameter of $d = 3$ because the absence or addition of transpositions will have the most effect when the depth parameter is the highest. Furthermore, we chose 1100 simulations as this is the highest value in the range of simulations per time step considered in our experiments.

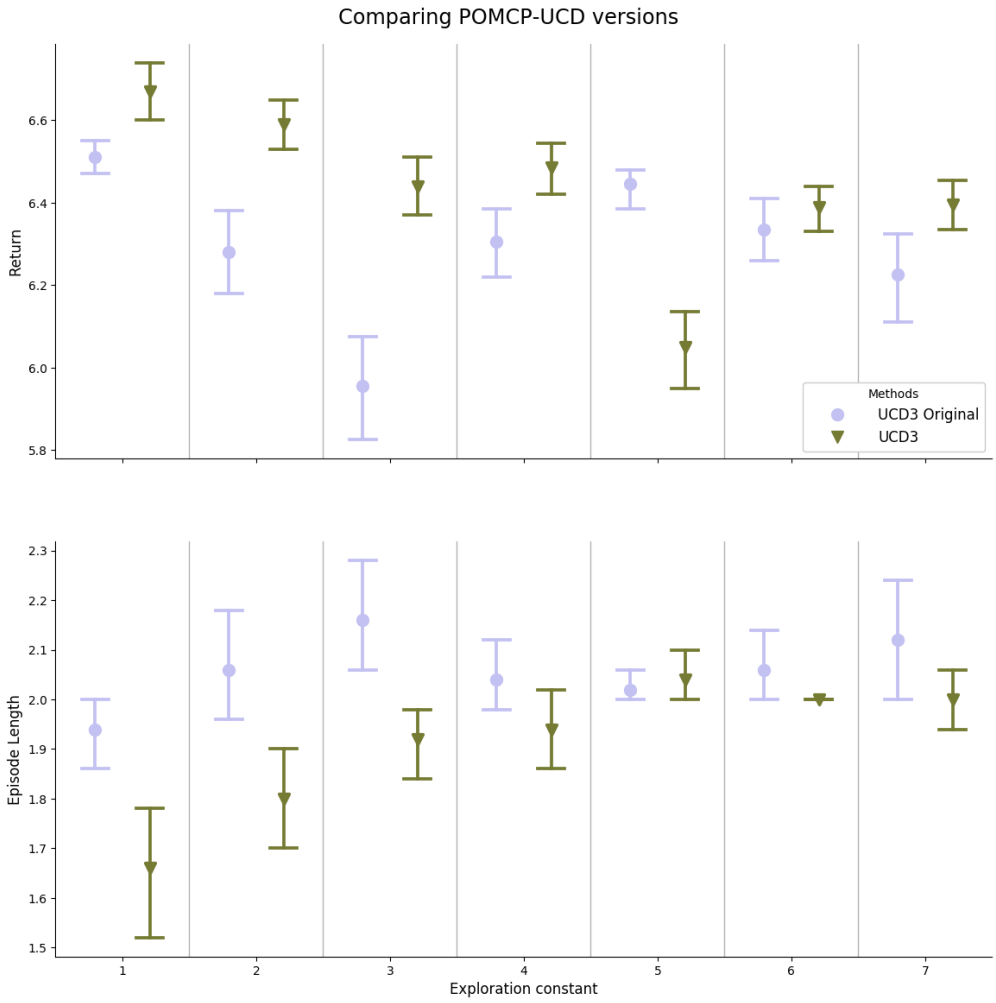


Figure 8.2: Comparison between the two POMCP-UCD versions with depth parameter $d = 3$. Shown are the averages and 95% confidence intervals over 50 runs for exploration constants in the range $c = \{1, 2, 3, 4, 5, 6, 7\}$ and 1100 simulations per time step.

Out of the 7 exploration constants, only $c = 5$ results in a worse performance of POMCP-UCD3 compared to the POMCP-UCD3 Original. For all other exploration constants POMCP-UCD3 clearly outperforms the Original version. The same holds true for the other scenarios in appendix E. Only for $c = 7$ on scenario 2 and $c = 4$ on scenario 1 does the Original version slightly outperform the POMCP-UCD3 algorithm.

8.2.2 POMCP-UCD Parametrizations

We now compare the different UCD parametrizations possible for a budget of $K = 3$. To do so we compare the different parametrizations on a fixed scenario and run the methods with the following exploration constants: $c = \{1, 2, 3, 4, 5, 6, 7\}$. Figure 8.3 shows the results of the maximal performing exploration constant c for depth parametrizations $d = \{0, 1, 2, 3\}$ as a function of simulations per time step. Shown are 50 runs with the markers denoting the averages. The UCD results are compared to the upper bound and average received from the 5040 greedy policies.

Recall that the depth parametrization determines the amount of transpositional information used for the exploration phases of the tree search. A value of $d = 0$ implies that no transpositional information is used and $d = 3$ implies that transpositions occurring 3 levels down the search DAG are considered. For $K = 3$ the action DAG is never deeper than 3 levels and therefore $d = 3$ is the largest possible depth parametrization for POMCP-UCD in our problem instance.

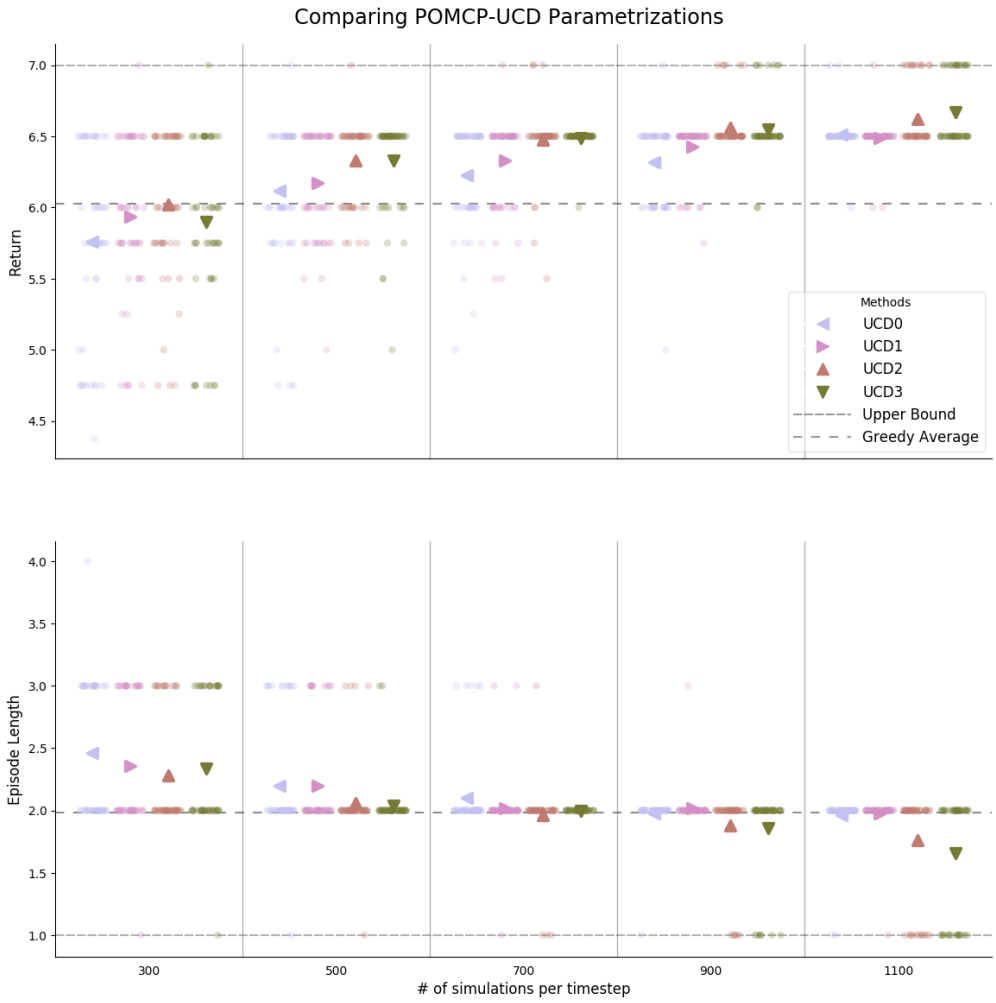


Figure 8.3: The max over exploration constants $c = \{1, 2, 3, 4, 5, 6, 7\}$ for all possible depth parametrizations $d = \{0, 1, 2, 3\}$ of POMCP-UCD. Shown are 50 runs with simulations per time step in the range of $\{300, 500, 700, 900, 1100\}$. The markers denote the averages over those 50 runs.

The figure seems to show that the parametrizations $d = 2$ and $d = 3$ perform very similarly. This also seems to be somewhat consistent for the other scenarios in appendix E. Overall, there seems to be a trend of higher parametrizations performing better. Parametrizations $d = 2$ and $d = 3$ seem to outperform $d = 1$ and $d = 1$ seems to slightly outperform $d = 0$. The differences between $d = 2$ and $d = 3$ are small. However, $d = 3$ seems to scale slightly better with increasing simulations per time step (although the differences are likely too small to be significant).

8.2.3 All Methods

The next experiment compares the POMCP-UCD with $d = 3$ algorithm with the other POMCP-based methods of this thesis. Figure 8.4 again shows the maximum performing exploration constant c for the different methods as a function of simulations per time step. Shown are 50 runs with the markers denoting the averages. The results are compared to the upper bound and average received from the 5040 possible greedy policies.

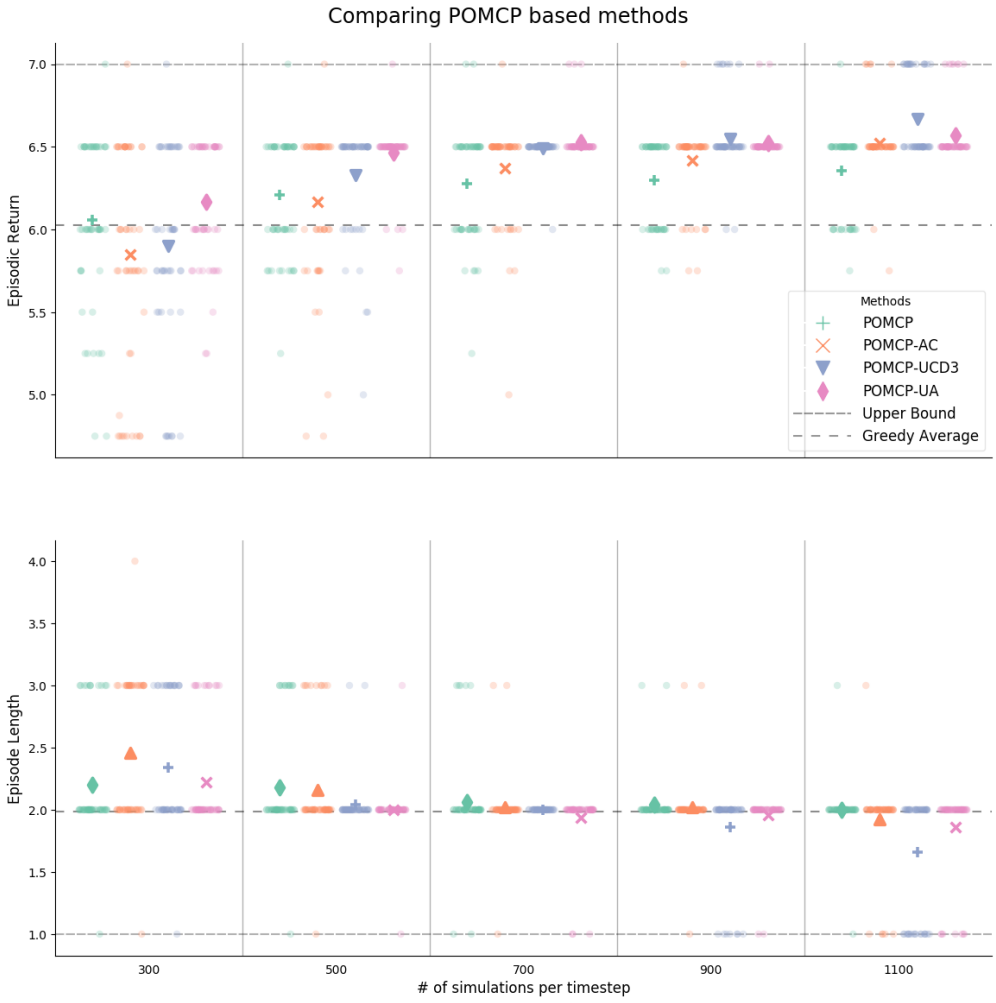


Figure 8.4: The max over exploration constants $c = \{1, 2, 3, 4, 5, 6, 7\}$ for all POMCP-based methods. Shown are 50 runs with simulations per time step in the range of $\{300, 500, 700, 900, 1100\}$. The markers denote the averages over those 50 runs.

The figure shows the following trends that seem to be consistent across all

the scenarios (for the other scenarios see appendix E). The POMCP-UA method seems to perform the best if there are only a small amount of simulations available per time step. However, this method seems to under-perform when the simulations per time step is increased.

On the other hand, the POMCP-UCD3 and POMCP-AC methods seem to under-perform for small amount of simulations per time step. However, both seem to scale the best when simulations per time step (search time) is increased. This results in the POMCP-UCD3 method outperforming all other methods for 1100 simulations per time step.

The POMCP method seems to follow the characteristics of the POMCP-UA method in terms of sample efficiency and scaling compared to the other two methods. POMCP has an initial decent performance for 300 simulations per time step (although outperformed by POMCP-UA) and seems to scale slower than POMCP-UCD3 and POMCP-AC (and similar to POMCP-UA).

In terms of episodic return, all the POMCP-based methods eventually (significantly) outperform the average greedy performance for 1100 simulations per time step (for all scenarios). They are also reaching the greedy based upper bound on some of the 50 runs. One more thing to note is that for every scenario, the POMCP-UA and POMCP-UCD3 methods outperformed the average greedy on all 50 of those runs (except for a single run of POMCP-UCD3 in scenario 3).

In terms of episode length, the POMCP-based methods only outperform the average greedy performance at 1100 simulations per time step for scenario 4. For all other scenarios, the POMCP-based methods under-perform compared to the average greedy policy for this metric.

8.3 Baseline Comparison

The POMCP-UCD3 method seemed to perform the best of all the POMCP-based methods when increasing the simulations per time step. In this experiment, we will increase the simulations per time step significantly to investigate how well the POMCP-based method can perform compared to the greedy policies. We run the POMCP-UCD3 method with $c = 7$ and 30000 simulations per time step a single time on 100 different scenarios. We compare this with the greedy upper bound, obtained by running all 5040 greedy policies and taking the best performing per scenario. So, the greedy upper bound is specific to the scenario. Figure 8.5 shows the percentage of scenarios in which the POMCP-UCD3 method under-performed/over-performed in terms of return and episode length.

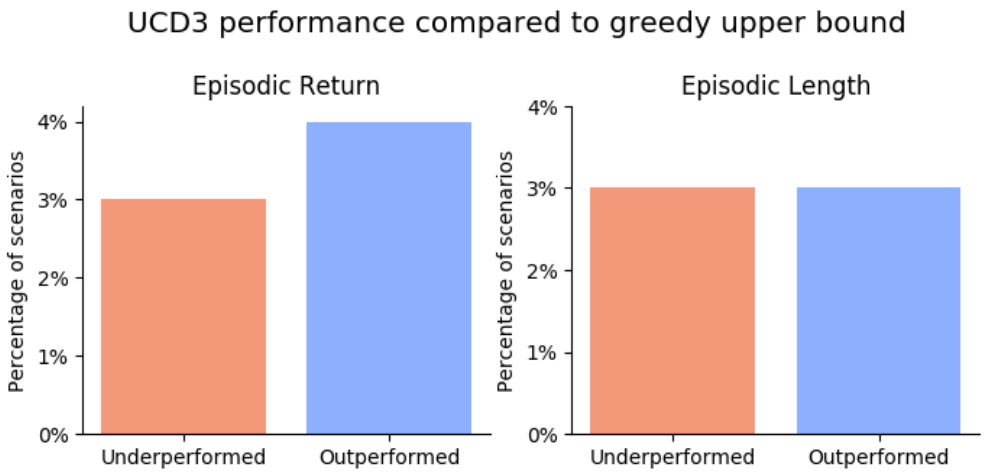


Figure 8.5: A single run of POMCP-UCD with depth parameter $d = 3$, $c = 7$ and 30000 simulations per time step compared to the greedy upper bound on 100 scenarios. The bars in blue denote the percentage of scenarios in which the POMCP-UCD3 method outperformed the greedy upper bound. The bars in red denote when it under-performed.

We see that the POMCP-UCD3 outperforms the greedy upper bound in terms of return on 4 out of the 100 scenarios. It also under-performs on 3 out of 100 scenarios. In terms of episode length, POMCP-UCD3 both under-performs and outperforms on 3 of 100 scenarios. On all other scenarios, the POMCP-UCD3 equalled the performance of the greedy upper bound.

In 3 of the 4 scenarios that outperformed in terms of return, POMCP-UCD3 also outperformed in terms of episode length. The other 1 scenario in which POMCP-UCD3 outperformed in terms of return, actually under-performed in terms of episode length. The other two scenarios in which POMCP-UCD3 under-performed in terms of episode length, it also under-performed in terms of return. Lastly, the other 1 scenario in which it under-performed in terms of return, it performed equal in terms of episode length. A summary of this can be found in table 8.1.

	Scenarios
Return & **Length**	5, 69, 100
Return & ***Length***	10, 50
Return & Length	27
Return & ***Length***	74

Table 8.1: Taxonomy of scenarios that under-performed/outperformed in at least one of the metrics as shown in figure 8.5. The color blue (or underline) indicates that POMCP-UCD3 outperformed the greedy upper bound, red (or italics) indicates it under-performed and grey indicates it performed equally.

8.3.1 Policy Evaluation

It turns out that in 3 of the 100 scenarios in which POMCP-UCD3 outperformed in terms of both metrics, it was due to the investigation of a particular licit node together with an illicit one. Figure 8.6 shows scenario 5 in which this occurred. Shown is a sub-graph of the full transactional network induced by the shortest paths between any two illicit nodes.

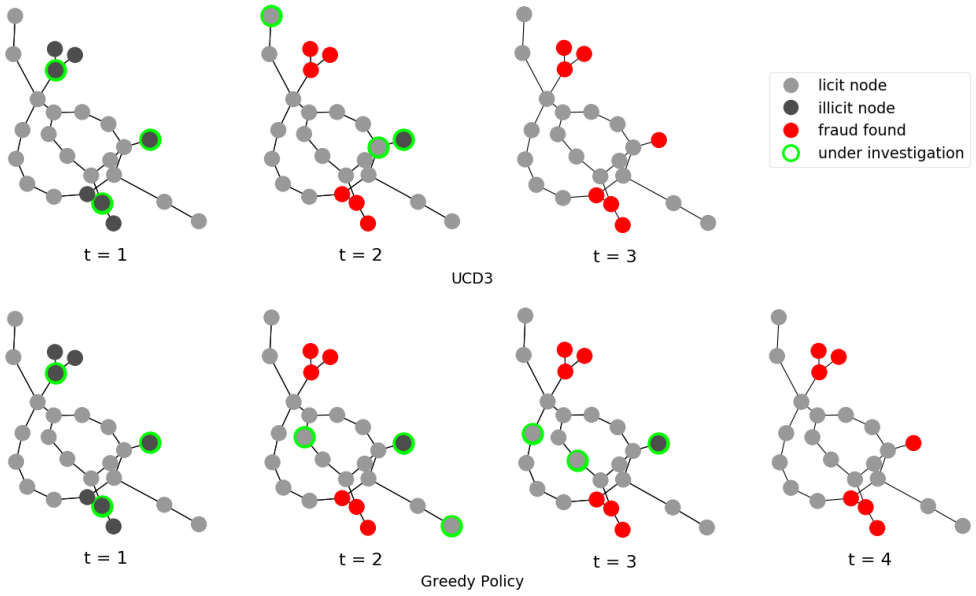


Figure 8.6: Policy comparison between POMCP-UCD3 (with $c = 7$ and 30000 simulations per time step) and the greedy upper bound policy. Shown is a scenario in which the POMCP-UCD3 policy outperforms the greedy one by selecting an illicit adjacent licit node in step 2.

We can see here that both policies choose the optimal first action, resulting in finding fraud in 6 out of the 7 illicit nodes in this network. However, in the second

time step, the POMCP-UCD3 policy investigates a licit node that is neighbouring the single illicit node left to find. Investigating this neighbouring licit node results in observing the features of this node, which increase the probability of finding fraud in the illicit node by a small amount (see section 6.2 for details on how this works). The greedy policies investigate licit nodes randomly. In most scenarios, the small increase in probability by investigating an illicit adjacent licit node will not change the outcome of an investigation. However, in 3 out of 100 scenarios this resulted in the POMCP-UCD3 policy outperforming the greedy upper bound.

Another interesting scenario is scenario 50 in which the POMCP-UCD3 underperformed in terms of both metrics. In this case, it turned out there was a particular greedy policy that performed optimally in this scenario, but sub-optimally on most other scenarios. Figure 8.7 shows both policies in this scenario.

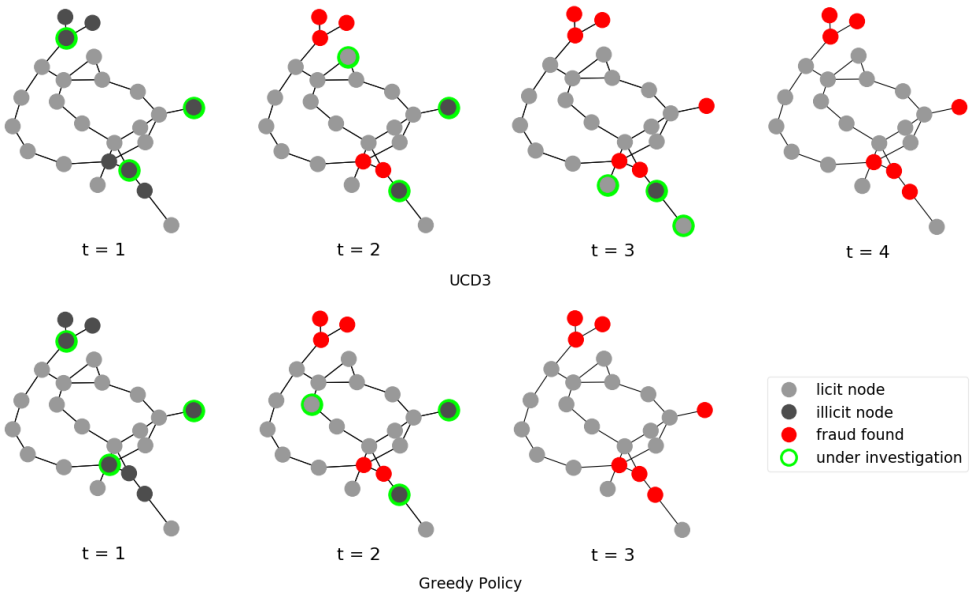


Figure 8.7: Policy comparison between POMCP-UCD3 (with $c = 7$ and 30000 simulation per time-step) and the greedy upper bound policy. Shown is a scenario in which the POMCP-UCD3 policy underperforms due to the specifics of this one scenario.

In this scenario, the optimal greedy policy is one that does not investigate one of the illicit nodes (the lowest one in the graph) in the first step. In this particular scenario, the first draw of the southern-most illicit node is more favourable than the second draw. Investigating this node for the first time once its neighbouring illicit node has been found (step 2 of the greedy policy) will result in finding fraud. Due to the specifics of our model and its parameters, investigating this node when its neighbouring illicit node hasn't been found yet (step 1 UCD3 policy) will not. Its neighbour being already found increases the probability of

labeling by a small amount that's large enough to flip the result from not found to found for the first draw. Because the second draw in this scenario happens to be unfavourable, this small amount is not enough to flip the results (step 2 UCD3 policy) for the second draw, requiring a third investigation into this node before it is found (step 3 UCD3 policy).

So, the greedy policy performed better because its first action was a sub-optimal one which happened to be optimal in this particular scenario. This shows that the POMCP-UCD3 method underperformed the greedy upper bound even though it found the 'correct' optimal action (optimal in most other scenarios). It is worth noting that in scenario 10 (the other scenario in which POMCP-UCD3 underperformed in both metrics) the underperforming seemed to be due to POMCP-UCD3 not finding the optimal sequence of actions it could have potentially found (since these would be also optimal in most other scenarios).

Chapter 9

Discussion

This chapter will discuss the results from chapter 8. It will go through the results section by section from beginning to end as presented in chapter 8.

9.1 Comparing POMCP Implementations

In the results from this section (figure 8.1) we found that the regular POMCP implementation scaled rather poorly in terms of memory footprint, especially as the budget K was increased. The memory-efficient implementation, on the other hand, kept a steady memory footprint as both simulations per time step and budget K were increased.

Barely increasing footprint for memory-efficient implementation

In the memory-efficient implementation, nodes are permanently stored in memory when they are traversed and temporarily when selecting a new action to take. We would expect the permanently stored memory to be mostly a function of simulations per time step (as more simulations means more of the search tree is traversed). Moreover, we would expect the temporarily stored nodes to be a function of the budget K (as a larger budget means more uninitialized nodes need to be temporarily initialized).

In our problem, the initial values are the same for all nodes: $\mu_{init} = 0, n_{init} = 0$. This means we don't actually have to initialize new nodes during the selection of an action, instead, we can treat all uninitialized nodes as the same and pick one at random if $u_{init} = \mu_{init} + c\sqrt{\frac{\log n(h)}{n_{init}}}$ is larger than all other $u(ha)$. Our implementation exploits this and does not temporarily initialize the uninitialized nodes (other than a single dummy node, representing all uninitialized ones).

For this reason, we would expect the memory footprint of this implementation to increase with simulations per time step and stay (relatively) unchanged with increasing budget K. However, in our results we found the opposite to occur:

the memory footprint remained relatively steady with simulations per time step and increased slightly more with increasing budget K. These differences were relatively small though. Perhaps the scale of the experiments (in terms of K and simulations per time step) was too small to observe the trends we would expect based on the above reasoning.

Memory usage/Execution time trade-off

It was mentioned in section 6.1.2 that the memory-efficient implementation performed a trade-off between memory usage and execution time. This was because it traded off storing all action nodes in memory (memory usage) for temporarily initializing those nodes when selecting a next action (execution time).

In the case of a uniform initial value across all nodes (for instance $\mu_{init} = 0, n_{init} = 0$), it is not necessary to initialize the uninitialized nodes during action selection (because they are all the same). In this case, there is no longer a trade-off between memory and time. Instead, the memory-efficient implementation is both quicker and more memory-efficient. If the initial values were unique for every (or most) node(s), creating this single 'dummy' node representing all uninitialized ones is of course not possible.

So, if this uniform initial value is not exploited, there could be a significant increase in execution time when using the memory-efficient implementation. This is because initializing (for instance) a million nodes every time you select an action could be very time-consuming. This only becomes worse if there is a complicated heuristic/offline solution used for the initial values.

It is worth noting that the regular POMCP implementation isn't especially time-efficient either. Since looping through (for instance) millions of nodes every action selection and initializing millions of nodes every expansion phase is time-consuming as well. So it is not immediately clear that the regular POMCP implementation would be much faster in those situations.

Other literature

We thought this implementation was worth noting since we did not find mention of this in the literature. In fact, in [20] they state that in their initial experiments the POMCP solver ran out of memory on a 30 node graph. However, their experiments ran on a machine with 48 GB of memory, with a typical budget size of $K = \{2, 3, 4, 5, 6\}$. In our experiments, we showed that the memory-efficient POMCP implementation could handle 50 node networks up to $K = 5$ without running out of memory.

9.2 Exploiting Combinatorial Structure

This section was designed to compare the methods in this thesis that exploit the combinatorial structure with various degrees.

9.2.1 POMCP-UCD Implementation

In this experiment, we compared the POMCP-UCD3 algorithm with one that did not exploit prior knowledge (POMCP-UCD3 Original). The POMCP-UCD version looks for and adds transpositions before choosing an action in the selection phase. The Original version only adds transpositions it encounters as a result of choosing an action.

POMCP-UCD exploits the fact that in our domain all possible transpositions are known before encountering them. Therefore, intuitively we would expect this version to perform better than the Original one. The results showed this to be the case with the exceptions of exploration constants $c = 5$ in scenario 4, $c = 7$ in scenario 2 and $c = 4$ in scenario 1.

Repeated experiments did not show any consistency of these exceptions across scenarios, exploration constants or simulations per time step. Therefore, it seems the exceptions could be due to the variance inherent to the POMCP-based approach.

9.2.2 POMCP-UCD Parametrizations

This experiment compared the possible parametrizations of the POMCP-UCD approach as a function of simulations per time step. We found there seemed to be a trend that higher parametrizations performed better, with $d = 2$ and $d = 3$ performing very similarly across scenarios.

Understanding the exploration depth parameter d

The higher the exploration depth parameter d, the more transpositional information is taken into account for the exploration of the search DAG. However, this can have the opposite effect on the exploitation of the search DAG. This is because the adapted score μ_d in the UCD framework uses the regular count n rather than the adapted count n_d, which is used for the exploration factor $\sqrt{\frac{\log\left(\sum_{f \in b(e)} n_d(f)\right)}{n_d(e)}}$. An example showing this can be found in figure 9.1.

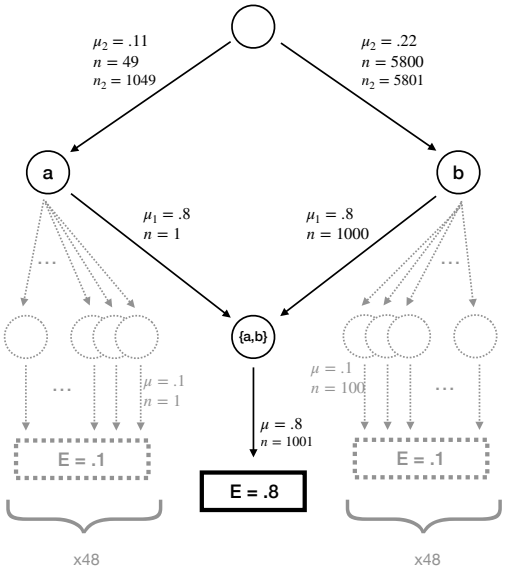


Figure 9.1: A scenario that showcases the difference in exploration between a small and large exploration depth in the UCD framework.

In this example, we see that even though the actions a and b are completely symmetric, the adapted score μ_2 values action b twice as much as action a. If exploration depth d is 0, the exploration will be based on $n_0 = n$. In this case, the exploration term will force the algorithm to explore action a more than action b, which might lead to equalization in adapted scores $\mu_2(a)$ and $\mu_2(b)$.

However, if exploration depth d equals 2 (the maximum in this search DAG), exploration will be based on n_2 rather than n. The adapted count $n_2(a)$ for action a is about 20 times larger than the regular count $n(a)$, even though the same does not hold for action b: $n_2(b) = 5801$ and $n(b) = 5800$. This means that in order to achieve the same level of exploration as $d = 0$, in the $d = 2$ case the exploration constant would have to be about 5 times larger.

So, with exploration constant kept (relatively) constant, a higher exploration depth parameter d could potentially lead to less exploration and more exploitation. This could be one of the causes for the higher depth parameters $d = 2$ and $d = 3$ performing better than the lower ones $d = 0$ and $d = 1$.

9.2.3 All Methods

This experiment compared all the different POMCP-based methods as a function of simulations per time step. The first trend we found was that the POMCP-UA method seemed to perform the best with low simulations per time step.

Update-all

The POMCP-UA method is the method that identically updates all possible ascent paths from a leaf node to the root of an action-DAG in the backpropagation phase. Furthermore, this method initializes in the action-DAG all possible transpositions from the current leaf node back to the root. In this way, the method exploits both the prior knowledge (what transpositions are possible) and sampled knowledge (by updating every possible ascend path identically) to a higher degree than the other methods in this thesis. Intuitively, it seems reasonable that when sampled knowledge is sparse (low number of simulations per time step), this method, therefore, performs the best. However, it is not necessarily clear to us why it is outperformed by the POMCP-UCD3 method as simulations per time step is increased.

Action chains and UCD3

One interesting thing we found in the results is that the POMCP-AC method behaved similar to the POMCP-UCD3 method. Both performed poorly for low simulations per time step, but rapidly increased in performance as the simulations per time step increased.

As discussed in section 9.2.2, the high depth parameter $d = 3$ of the POMCP-UCD approach can cause it to asymmetrically value one sub-action over another, even if both sub-actions lead to completely symmetric parts of the search space. This behaviour was observed during some of the experiments in our problem scenario. Interestingly, the POMCP-AC method behaves similarly.

The POMCP-AC method ignores the symmetries of the combinatorial action space. It, therefore, undervalues certain sub-actions compared to the value they could have if these symmetries were taken into account. However, this undervaluing of those sub-actions leads to more exploitation of areas of the search tree/DAG that show promise early on.

Both the POMCP-AC and POMCP-UCD3 methods share this behaviour. Intuitively, it also makes sense that this behaviour is not optimal if there is only a small amount of simulations per time step. In this case, the methods can more easily over-fit on a part of the search tree/DAG that was found to be locally optimal. However, this balance of more exploitation versus exploration could

be a more optimal balance for higher simulations per time step. This could potentially explain the similar behaviour of POMCP-AC and POMCP-UCD3.

Note that it is not clear to us that this simply is a consequence of exploration versus exploitation, as it is not clear to us whether the POMCP-UA method explores or exploits more compared to the POMCP-AC and POMCP-UCD3 methods. Nor is it clear why the POMCP method seems to behave similar in these regards as the POMCP-UA method.

Average greedy performance

Another thing we found is that in terms of the episodic return the POMCP-based methods consistently outperformed the average greedy performance for 1100 simulations per time step. The average greedy performance can be viewed as the performance one can (on average) expect from a randomly picked greedy policy (randomly picked with respect to the investigation dynamics). In a way this can be interpreted as the average performance one can expect of a completely investigation-dynamic agnostic greedy policy. Therefore, one could make the case that a sequential decision making approach is already worth consideration if it can outperform this average performance.

Episodic Return vs Episode Length

Another interesting thing to note about the results is that the POMCP-based methods did not consistently outperform the average greedy performance in terms of episode length. This could partially be due to the fact that the POMCP-based methods value their sampled actions during the lookahead search based on the return, not the episode length. In this way, it is the return that the POMCP-based methods try to directly optimize, not the episode length. Perhaps similar performance in terms of episode length could be achieved if sampled actions during the POMCP-based methods were valued according to that metric.

9.3 Baseline Comparison

In this experiment, we increased the simulations per time step of the POMCP-UCD3 method to 30000, in order to investigate how it compares to the greedy upper bounds. Before running this experiment, we expected the greedy class of policies to contain the most optimal policies possible. We expected this because we assumed the licit nodes that are investigated after the illicit ones would have no influence on rewards (since investigating a licit node will always give zero reward).

Furthermore, due to the problem parameters used in this thesis, the effect an investigation of a licit node has on the labeling of a neighbouring illicit node is relatively small. This means it will never be optimal to investigate a licit node before an illicit one. This is why the greedy class of policies contains every possible ordering of the 7 illicit nodes, together with randomly sampled licit nodes after those. Note that including every possible ordering of licit nodes in the class of greedy policies would not be possible from a practical standpoint, even if we did assume the licit ordering would influence the rewards in any significant way.

The power of sequential decision making approaches

So, before running this experiment, we expected at best the POMCP-UCD3 method to reach the scenario-specific upper bounds. However, we found that in 4 out of 100 scenarios it actually outperformed those upper bounds. We found that in 3 out of 100 scenarios the situation occurred in which investigating a specific licit node together with the last illicit node changed the outcome of the labeling in that time step.

This demonstrates the potential power of modelling your problem as a sequential decision making problem and optimizing it. The sequential decision making approach will optimize the model as it is given. This can sometimes lead to the discovery of unexpected behaviour of the model. If the behaviour is very unrealistic, this could indicate an issue with the current description of the model. However, if the behaviour is not necessarily unrealistic, interesting new heuristics could be learned from this behaviour.

POMCP-based methods versus baseline

All in all the POMCP-based method outperformed the baseline in terms of episodic return (better in 4% and worse in 3%) and equal in terms of episode length. This is impressive given that the baseline is an upper bound on the greedy policies. We demonstrated that a *single* POMCP-based method could outperform/ perform equal to the scenario-specific best-performing policy out of an entire class of greedy policies. So, even in the unrealistic situation in which one could beforehand pick the best greedy policy specific to the scenario that was about to occur, the POMCP-based method still performed equal or better.

Under-performance

In 3 out of 100 scenarios the POMCP-UCD3 method under-performed in terms of return. We discovered that in 1 of those scenarios it under-performing was due to it selecting the move that would be optimal in most other scenarios. However, in the other 2 scenarios, this was not so clear. This could indicate

that in those other scenarios the POMCP-UCD3 method could potentially still reach the greedy upper bound (or surpass it) if the simulations per time step is increased further.

Part V

Conclusion

Chapter 10

Conclusion & Future Work

The main objective of this thesis is to frame the anti-money laundering process as a sequential decision making process and optimize it. This chapter will list the main contributions of this thesis in relation to the research questions stated in the introduction. It will also contain a recommendation for ING and a discussion on the limitations and interesting future research topics.

10.1 Conclusion

The main research question of this thesis is *Can Anti-Money Laundering practices be improved by sequential decision making algorithms?* The first contribution of this thesis to answering this question is the defining and designing of AML as a Partially Observable Markov Decision Process.

Model

As far as we know, no prior research has tried to formalize the AML process as a sequential decision making process. The first contribution of this thesis is therefore to formally define the AML process as a POMDP. The challenge we encountered was that intuitively, in AML, the observations (investigations) depend on the previously received observations (previous investigations). However, in a POMDP, observations can only depend on the states and actions. We overcame this by adding the observations to the definition of our state space. In this way, we answered the first research question: *Is it possible to model the AML decision making process in the POMDP framework?*

Moreover, to our knowledge, our thesis is the first to design and implement a generative model of the AML investigation process. The aim was to design a model simple enough to be implemented as part of this thesis, yet complex enough to produce interesting results that are based in reality. For this purpose, we designed a model that uses the fraction of accurate feature values as a measure of knowledge on which the labeling of nodes as illicit/licit is based. This measure

can be increased through investigations of a node and we use the risk score to guide the investigations to a node's surroundings.

POMDP solution approaches

One of the main challenges of this thesis was to overcome the challenges associated with the scale of anti-money laundering POMDP. This thesis uses an approximate, online and model-free approach called the Partially Observable Monte-Carlo Planning approach to overcome the challenges associated with the large state space.

However, this approach suffers a major memory bottleneck for problems with large action spaces such as ours. For this reason, we design a memory-efficient implementation of the POMCP algorithm that does not suffer a large memory footprint for large action spaces. We discuss that this implementation contains a memory/time trade-off that should be favourable if the initial action values are not too complex to derive. In fact, we show in this thesis that if the initial values are the same for every action (we have zero prior knowledge on the values of actions), the implementation can be further optimized to achieve huge performance gains in terms of both memory *and* execution time.

Exploiting the action space structure

Another challenge of the AML POMDP as defined in this thesis is the large action space. The large size of the action space is due to the combinatorial nature of the actions in the AML POMDP (selecting subsets of nodes of a particular size out of a larger set). This challenge is reflected in the second research question *How can we exploit the combinatorial structure of the action space to improve the performance of POMDP solution approaches?* This thesis tries to answer this research question by proposing a number of different POMCP-based solution approaches that exploit this structure in various degrees.

The first method builds upon an idea from similar domains where the structure of the action space is exploited by partitioning a single action (selecting nodes) into a chain of sub-actions (selecting a single node). We propose a method called the POMCP with Action Chains (POMCP-AC) method that is an extension of the POMCP algorithm with the addition of action chains in the belief tree.

The limitation of this method is that it ignores the symmetries of the combinatorial structure. To exploit this we propose to add transpositions to the action chains turning them into what we call action-DAGs. This turns the belief tree into a belief Directed Acyclic Graph (DAG). Furthermore, we propose to exploit our prior knowledge of these symmetries by pro-actively adding transpositions to the action-DAGs.

Based on this, we propose the POMCP-UCD method that adapts the general UCD framework of Monte-Carlo Tree Search (MCTS) for Directed Acyclic Graphs to the action-DAGs of the belief DAG. This method exploits our prior knowledge of the combinatorial symmetries by pro-actively adding transpositions that directly inform the MCTS selection phase.

However, the UCD framework does not maximally exploit the transpositional information obtained during the Monte-Carlo sampling of the search space. We therefore also propose a method POMCP-UA that applies the update-all strategy for general DAGs to the action-DAGs in our belief DAG. We also design the POMCP-UA to exploit the prior knowledge of the combinatorial symmetries by adding all possible descent paths leading to the leaf nodes we traverse. This method exploits both the prior knowledge and sampled knowledge to a higher degree than any of the other methods in this thesis.

Empirical evaluation

In order to answer research question three, *How do sequential decision making approaches compare to methods that ignore the investigation dynamics?*, and part of the second research question, we perform several experiments on a relatively small transactional network.

In those experiments, we find that the method that exploits the combinatorial structure the most (POMCP-UA) outperforms all others if only a small amount of Monte-Carlo samples of the search space are drawn. This perhaps suggests that maximally exploiting/sharing the sampled information is most important if samples are rare.

However, we find that the POMCP-UCD method seems to scale better if we increase the lookahead search time per time step. This results in the POMCP-UCD method overtaking the performance of the POMCP-UA method for longer search times. Interestingly, the same scalability seems to be observed in the POMCP-AC method, which exploits the combinatorial structure relatively minimaly. This might suggest that maximally exploiting the known structure might suffer from sub-optimal exploration/exploitation of the search space in this domain.

We show that the above methods outperform the average greedy performance of the baseline for relatively small search times (in the order of up to 1000 simulations per time step). This might suggest that the POMCP-based methods can outperform the average performance of a method that is completely agnostic of the investigation dynamics in reasonable computation times.

Moreover, we show that increasing the computation time significantly (to the order of tens of thousands of simulations per time step) for the POMCP-UCD method can lead it to even slightly outperform the unrealistic greedy upper

bounds. This shows that this method converges to a policy that is better than the best you can expect from the baseline.

10.2 Limitations & Future Work

It is worth adding some nuance here on the implications of the results of this thesis on the real-life problem of anti-money laundering in financial institutions.

Model

The results of this thesis were obtained using the model described in chapter 4. The implications of the results and how they would translate to the real-life problem is completely dependent on the quality of the model.

It is not necessarily the case that we need a very high fidelity, realistic model before the results can be translated to the real-life problem. It could be for instance that a simple model contains a 'realistic' property or behaviour that manifests in a certain behaviour in the policy. This behaviour in the policy can then lead to new insights that can improve upon current methods or help design new ones.

We are not experts in the area of anti-money laundering investigations or even in the area of money laundering itself. We have designed the model to be simple but realistic to the best of our ability. However, we refer to other experts to conclude whether our model translates in any way to the real-world problem.

One interesting area for future work could therefore be to improve the model. There are a number of different assumptions made to simplify the model for this thesis. One could expand on the complexity of the model by discarding certain assumptions such as a static graph, a graph only containing transactions, the fixed-size time steps, the fixed budget size, using binary labels to denote fraud, using only a single node type and/or the assumption that flagged nodes are always investigated. Another approach could be to keep the simplicity of the model and use expert knowledge gathered from the specialized investigators to fine-tune the realism of the model.

Risk Score

Another simplification made in our model is that the risk score is 100% accurate. Because the generative model depends on the risk score, this also means we assume the model to be 100% accurate. Some preliminary testing showed that if the risk score is erroneous (and therefore so is the model), the POMCP-based methods would converge to policies that far underperformed the greedy baselines. An interesting avenue of research could therefore be the incorporation

of the uncertainty on the model parameters/structure into the belief state of our POMDP. Bayesian reinforcement learning optimization methods would probably be an interesting topic for this scenario.

Methods & Empirical Evaluation

Additionally, it would be very interesting to get a better understanding of what causes the different behaviours of the POMCP-UA and POMCP-UCD approaches. The POMCP-UA method seemed to perform the best for small simulations per time step, but the POMCP-UCD method seemed to scale better when this was increased. In the discussion, we proposed that it could have something to do with the exploration/exploitation trade-off. It would be very interesting to investigate this further and perhaps look for ways to combine the strengths of both methods.

Additionally, it would be very interesting to further verify some of the trends that were observed during the empirical evaluation of the methods in this thesis. For one, if the difference in behaviour of the POMCP-UA and POMCP-UCD methods is indeed (partly) due to the exploration/exploitation trade-off, perhaps it could be interesting to increase the range of exploration constants with which the experiments were performed. Perhaps it could be possible to find a mapping from the behaviour of one method to another by increasing/decreasing the exploration constant.

Also, the behaviour of the methods with respect to the simulations per time step was only observed on a relatively small range of values. The behaviour with respect to this variable could be further investigated by increasing the range of simulations per time step for the experiments.

Dataset & Other Problem Domains

Our experiments relied on the AMLSim synthetic dataset to create a transaction network. This synthetic transaction network is not necessarily realistic. Furthermore, the way we designed the investigation model to interact with the graph is partly based on our knowledge of money laundering structures in the transaction network. Our knowledge was probably indirectly dependent on/similar to the knowledge used in the AMLSim simulator. In this way, there is always the danger of 'finding what you put in' when personally designing both the model and the solution method. The emphasis of the contributions of this thesis should therefore be focused on the proposed methods, that should be applicable to any problem with a similar action space structure.

One area of future research could be to investigate if the same results are observed on a different synthetic (or real) dataset. Moreover, the proposed methods should be applicable to any problem domain that exhibits the same combinatorial action space. It would be very interesting to verify that the trends and behaviours

of the methods encountered in this thesis apply to those other problem domains as well.

An example of such a domain could be a recommendation problem in which the action consists of a set of entities to recommend. For instance the recommendation of a set of advertisements or news feeds. Similarly, certain maintenance domains could be applicable as well, such as selecting a set of wind turbines to maintain in a wind farm. In addition, any problem in which the action consists of choosing a set of nodes from a graph should be relevant. An example of this is the problem of maximizing influence spread in social networks. Or more generally the framework of Combinatorial Multi-Armed Bandits (CMAB), in which one selects as an action a *super arm* consisting of a set of *base arms*.

10.3 Recommendations

The contributions of this thesis were mainly focused on the development of algorithms and architectures that deal with the challenges associated with the anti-money laundering POMDP. However, the relevance of those approaches to the real anti-money laundering process inside a financial institution such as ING, is heavily dependent on the quality of the model. If a financial institution such as ING decides to further invest in a sequential decision making approach, we would advise to:

1. Improve the quality of the model. This could be by adding more complexity or evaluating its realism. The latter one especially is highly tied to better understanding the investigation process that is a major part of the anti-money laundering process.

2. Study approaches to deal with an inaccurate risk score (such as Bayesian RL). In reality, the risk-scoring algorithm will not be flawless. This results in inaccuracies in the model which has to be taken into account to avoid the sequential decision making approach converging to a solution worse than the baseline.

3. Use the sequential decision making solutions to create a better understanding of the process and how to optimize it. It is unlikely that in the near future an anti-money laundering model will capture all the complexities of the real problem. So, rather than looking for a black box solution that optimizes the model, it would be more beneficial to use the sequential decision making approaches to find/test heuristics and improve the understanding of the investigation process.

Bibliography

[1] "United nations office on drugs and crime," https://www.unodc.org/unodc/en/money-laundering/globalization.html, accessed: 11-09-2020.

[2] Z. Chen, L. D. V. Khoa, E. N. Teoh, A. Nazir, E. K. Karuppiah, and K. S. Lam, "Machine learning techniques for anti-money laundering (AML) solutions in suspicious transaction detection: a review," *Knowl. Inf. Syst.*, vol. 57, no. 2, pp. 245–285, 2018. [Online]. Available: https://doi.org/10.1007/s10115-017-1144-z

[3] M. Weber, G. Domeniconi, J. Chen, D. K. I. Weidele, C. Bellei, T. Robinson, and C. E. Leiserson, "Anti-money laundering in bitcoin: Experimenting with graph convolutional networks for financial forensics," *CoRR*, vol. abs/1908.02591, 2019. [Online]. Available: http://arxiv.org/abs/1908.02591

[4] M. Weber, J. Chen, T. Suzumura, A. Pareja, T. Ma, H. Kanezashi, T. Kaler, C. E. Leiserson, and T. B. Schardl, "Scalable graph learning for anti-money laundering: A first look," *CoRR*, vol. abs/1812.00076, 2018. [Online]. Available: http://arxiv.org/abs/1812.00076

[5] A. Pareja, G. Domeniconi, J. Chen, T. Ma, T. Suzumura, H. Kanezashi, T. Kaler, T. B. Schardl, and C. E. Leiserson, "Evolvegcn: Evolving graph convolutional networks for dynamic graphs," in *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 2020, pp. 5363–5370. [Online]. Available: https://aaai.org/ojs/index.php/AAAI/article/view/5984

[6] A. F. Colladon and E. Remondi, "Using social network analysis to prevent money laundering," *Expert Syst. Appl.*, vol. 67, pp. 49–58, 2017. [Online]. Available: https://doi.org/10.1016/j.eswa.2016.09.029

[7] B. Settles, "Active learning literature survey," University of Wisconsin-Madison Department of Computer Sciences, Tech. Rep., 2009.

[8] J. Pineau, G. J. Gordon, and S. Thrun, "Anytime point-based approximations for large pomdps," *J. Artif. Intell. Res.*, vol. 27, pp. 335–380, 2006. [Online]. Available: https://doi.org/10.1613/jair.2078

[9] D. P. Bertsekas, *Dynamic programming and optimal control, 3rd Edition.* Athena Scientific, 2005. [Online]. Available: https://www.worldcat.org/oclc/314894080

[10] C. H. Papadimitriou and J. N. Tsitsiklis, "The complexity of markov decision processes," *Math. Oper. Res.*, vol. 12, no. 3, pp. 441–450, 1987. [Online]. Available: https://doi.org/10.1287/moor.12.3.441

[11] O. Madani, S. Hanks, and A. Condon, "On the undecidability of probabilistic planning and infinite-horizon partially observable markov decision problems," in *Proceedings of the Sixteenth National Conference on Artificial Intelligence and Eleventh Conference on Innovative Applications of Artificial Intelligence, July 18-22, 1999, Orlando, Florida, USA*, 1999, pp. 541–548. [Online]. Available: http://www.aaai.org/Library/AAAI/1999/aaai99-077.php

[12] S. Ross, J. Pineau, S. Paquet, and B. Chaib-draa, "Online planning algorithms for pomdps," *J. Artif. Intell. Res.*, vol. 32, pp. 663–704, 2008. [Online]. Available: https://doi.org/10.1613/jair.2567

[13] R. Coulom, "Efficient selectivity and backup operators in monte-carlo tree search," in *Computers and Games, 5th International Conference, CG 2006, Turin, Italy, May 29-31, 2006. Revised Papers*, ser. Lecture Notes in Computer Science, H. J. van den Herik, P. Ciancarini, and H. H. L. M. Donkers, Eds., vol. 4630. Springer, 2006, pp. 72–83. [Online]. Available: https://doi.org/10.1007/978-3-540-75538-8_7

[14] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck, "Monte-carlo tree search: A new framework for game AI," in *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference, October 22-24, 2008, Stanford, California, USA*, 2008. [Online]. Available: http://www.aaai.org/Library/AIIDE/2008/aiide08-036.php

[15] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *Machine Learning: ECML 2006, 17th European Conference on Machine Learning, Berlin, Germany, September 18-22, 2006, Proceedings*, 2006, pp. 282–293. [Online]. Available: https://doi.org/10.1007/11871842_29

[16] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Mach. Learn.*, vol. 47, no. 2-3, pp. 235–256, 2002. [Online]. Available: https://doi.org/10.1023/A:1013689704352

[17] D. Silver and J. Veness, "Monte-carlo planning in large pomdps," in *Advances in Neural Information Processing Systems 23: 24th Annual Conference on Neural Information Processing Systems 2010. Proceedings of a meeting held 6-9 December 2010, Vancouver, British Columbia, Canada*,

2010, pp. 2164–2172. [Online]. Available: http://papers.nips.cc/paper/4031-monte-carlo-planning-in-large-pomdps

[18] Z. N. Sunberg and M. J. Kochenderfer, "Online algorithms for pomdps with continuous state, action, and observation spaces," in *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling, ICAPS 2018, Delft, The Netherlands, June 24-29, 2018*, 2018, pp. 259–263. [Online]. Available: https://aaai.org/ocs/index.php/ICAPS/ICAPS18/paper/view/17734

[19] J. Scharpff, M. T. J. Spaan, L. Volker, and M. de Weerdt, "Planning under uncertainty for coordinating infrastructural maintenance," in *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling, ICAPS 2013, Rome, Italy, June 10-14, 2013*, 2013. [Online]. Available: http://www.aaai.org/ocs/index.php/ICAPS/ICAPS13/paper/view/6045

[20] A. Yadav, H. Chan, A. X. Jiang, H. Xu, E. Rice, and M. Tambe, "Using social networks to aid homeless shelters: Dynamic influence maximization under uncertainty - an extended version," *CoRR*, vol. abs/1602.00165, 2016. [Online]. Available: http://arxiv.org/abs/1602.00165

[21] B. E. Childs, J. H. Brodeur, and L. Kocsis, "Transpositions and move groups in monte carlo tree search," in *Proceedings of the 2008 IEEE Symposium on Computational Intelligence and Games, CIG 2009, Perth, Australia, 15-18 December, 2008*, P. Hingston and L. Barone, Eds. IEEE, 2008, pp. 389–395. [Online]. Available: https://doi.org/10.1109/CIG.2008.5035667

[22] A. Saffidine, T. Cazenave, and J. Méhat, "UCD : Upper confidence bound for rooted directed acyclic graphs," *Knowl. Based Syst.*, vol. 34, pp. 26–33, 2012. [Online]. Available: https://doi.org/10.1016/j.knosys.2011.11.014

[23] IBM. (2019) Amlsim wiki. [Online]. Available: https://github.com/IBM/AMLSim/wiki

[24] V. Krishnamurthy, *Partially observed Markov decision processes*. Cambridge University Press, 2016.

[25] L. D. Stone, *Theory of optimal search*. Elsevier, 1976, vol. 118.

[26] M. T. J. Spaan, "Partially observable markov decision processes," in *Reinforcement Learning*, 2012, pp. 387–414. [Online]. Available: https://doi.org/10.1007/978-3-642-27645-3_12

[27] E. J. Sondik, "The optimal control of partially observable markov processes." Stanford Univ Calif Stanford Electronics Labs, Tech. Rep., 1971.

[28] R. D. Smallwood and E. J. Sondik, "The optimal control of partially observable markov processes over a finite horizon," *Oper. Res.*, vol. 21, no. 5, pp. 1071–1088, 1973. [Online]. Available: https://doi.org/10.1287/opre.21.5.1071

[29] M. T. J. Spaan and N. A. Vlassis, "Perseus: Randomized point-based value iteration for pomdps," *J. Artif. Intell. Res.*, vol. 24, pp. 195–220, 2005. [Online]. Available: https://doi.org/10.1613/jair.1659

[30] S. Paquet, B. Chaib-draa, and S. Ross, "Hybrid pomdp algorithms," in *Proceedings of The Workshop on Multi-Agent Sequential Decision Making in Uncertain Domains (MSDM-06)*, 2006, pp. 133–147.

[31] S. Paquet, L. Tobin, and B. Chaib-draa, "An online POMDP algorithm for complex multiagent environments," in *4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005), July 25-29, 2005, Utrecht, The Netherlands*, 2005, pp. 970–977. [Online]. Available: https://doi.org/10.1145/1082473.1082620

[32] D. A. McAllester and S. P. Singh, "Approximate planning for factored pomdps using belief state simplification," in *UAI '99: Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence, Stockholm, Sweden, July 30 - August 1, 1999*, 1999, pp. 409–416. [Online]. Available: https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=193&proceeding_id=15

[33] D. P. Bertsekas and D. A. Castañón, "Rollout algorithms for stochastic scheduling problems," *J. Heuristics*, vol. 5, no. 1, pp. 89–108, 1999. [Online]. Available: https://doi.org/10.1023/A:1009634810396

[34] N. Ye, A. Somani, D. Hsu, and W. S. Lee, "DESPOT: online POMDP planning with regularization," *J. Artif. Intell. Res.*, vol. 58, pp. 231–266, 2017. [Online]. Available: https://doi.org/10.1613/jair.5328

[35] N. P. Garg, D. Hsu, and W. S. Lee, "Despot-alpha: Online POMDP planning with large state and observation spaces," in *Robotics: Science and Systems XV, University of Freiburg, Freiburg im Breisgau, Germany, June 22-26, 2019*, 2019. [Online]. Available: https://doi.org/10.15607/RSS.2019.XV.006

Appendix A

Assumptions

This appendix contains a summary off all the assumptions in our thesis.

Static Graph

We consider for now a static transactional graph consistent with taking a snapshot of the transactional network during a certain time window. For the most part the actions and observations received do not influence the dynamics/behaviour of the transactions. Solving for a static graph therefore seems like a good first step.

Binary labels

For now we will assume there are only binary labels that distinguish a licit from an illicit node. In the future, we might wish to use something more sophisticated like a continuous label ranging from 0 to 1 (describing how likely a node is to be illicit).

Only transactions

For now we will assume the only type of edge that explicitely exists in our data is a transactional edge.

Fixed size time-step

We assume the time it will take to investigate a node (and obtain an observation) is fixed, independent of whether the true label of the node is licit or illicit. In reality, this assumption does not hold. In real-life the investigations can take a variable amount of time which depends on the true label of the node.

Fixed size budget

In reality, the investigators might have to wait some time for some external information to become available. This is one of the causes of the variable time-step. However, this could also cause the investigator to start investigation on a new node whilst waiting for the information on the other. This complex behaviour could perhaps be modeled using a variable budget size together with variable time-steps. However, for now we assume a fixed budget size.

Nodes flagged will always be evaluated

We assume for now that nodes flagged (by taking an action a) will actually be investigated. In reality, the nodes we flag will only be suggestions to the team of investigators. They themselves decide which nodes to investigate (which could therefore differ from the ones flagged in a).

Immediate feedback

The end goal of our problem is to maximize the amount of Suspicious Activity Reports (SARs) sent to the public prosecution service. Because we assume fixed time steps we can model this by giving a reward everytime a node gets labeled as illicit. In reality this reward could be delayed.

Investigations also label suspicious neighbors

We will assume that all suspicious nodes encountered in the investigation process will receive SARs. This is not necessarily a realistic assumption, since in real life typically only the node that is formally under investigation can receive a SAR. In real life suspicious activity from a node v might be encountered during an investigation into node $a^i \in a_t$. In this case a new formal investigation would probably be launched into node v. In real life however, investigations don't take a fixed amount of time. The investigation into node v will probably proceed much more swiftly than it would normally because part of the investigation was already performed during the investigation into a^i. In the fixed time-step investigations we use, we will somewhat model this by including the SAR of the suspicious node v in the investigation into a^i.

Single node type

We assume for now all nodes are of the same type. In reality, different nodes correspond to different types of businesses which all have their own characteristics. A more sophisticated approach could take into account the different type of

businesses as the type of business most likely influences the probability of the it being illicit.

Observing features and fraud are independent

Observing features of a node (or its neighbours) is independent of observing the node as fraudulent or not. I am not sure how realistic this assumption is.

Observing a node is independent of the other nodes observed

Observing a node as fraudulent has no effect on observing other nodes as fraudulent in that time step. It might affect it in the next time step however.

Investigators don't falsely label licit nodes as illicit

We will assume that the probability of labeling a licit node as illicit is 0. In reality this might not be entirely realistic. However, it is not unreasonable to assume this probability is relatively low.

Our Risk Score is 100% accurate

Nodes that are actually fraudulent will always have higher risk scores than nodes that aren't. This isn't a very realistic assumption in the real life.

Appendix B

Similar Problem Domains

There are a number of existing problem domains/frameworks that are similar to ours.

B.1 Controlled Sensing

Controlled sensing (or active perception) is a special case of a POMDP where the decision-maker (controller) controls the sensing part (observations) of the POMDP but not the underlying MDP [24]. In controlled sensing, the controller controls the sensing by switching between sensors or sensing modes. Typically, in these types of problems an accurate sensor yields less noisy observations but is more expensive to use than an inaccurate one. Because the controller only receives the (noisy) observations and not the states, optimizing this process requires decision making under uncertainty and is nontrivial to solve. An example of a controlled sensing problem is radar scheduling for tracking a maneuvering target.

In [24], they define the controlled sensing problem as trying to minimize the following finite horizon cumulative cost function (as opposed to maximizing a reward function):

$$J_\pi = \mathbb{E}_\pi \left\{ \sum_{t=0}^{N-1} \left[c(s_t, a_t) + d(s_t, b_t, a_t) \right] | b_0 \right\} \tag{B.1}$$

where the expectation is over a policy π. $s_t \in S$, $a_t = \pi_t(b_t) \in A$ and $b_t \in \Pi(\mathcal{S})$ denote respectively the state, the sensor and the belief at time t. The instantaneous costs are divided in two types:

- Sensor usage cost $c(s_t, a_t)$: This denotes the instantaneous cost of using sensor a_t when in state s_t at time t.
- Sensor performance loss $d(s_t, b_t, a_t)$: This cost models the performance loss one has when using sensor a_t given the belief state b_t and actual state s_t. This cost will be larger the more uncertain you are about the state.

Usually there is a trade off between the two costs for any particular sensor. For instance, an accurate sensor will have high sensor usage cost but low sensor performance loss.

Examples of Sensor performance loss

The sensor performance loss is an indication of the uncertainty of the observer. As such, this cost should be zero at the vertices of the belief state simplex (where we are certain about being in a particular state) and largest at the centroid (where we are most uncertain). An example could be a piecewise linear cost:

$$d(s,b,a) = \begin{cases} 0 & \text{if } ||s-b||_\infty \leq \epsilon \\ \epsilon & \text{if } \epsilon \leq ||s-b||_\infty \leq 1-\epsilon \\ 1 & \text{if } ||s-b||_\infty \geq 1-\epsilon \end{cases} \tag{B.2}$$

where $\epsilon \in [0, 0.5]$. This cost divides the belief space into three regions: close, medium and far.

Similarity to our problem

Our problem domain is similar to the problem domain of controlled sensing in that our actions only control the sensing part of the POMDP. In the AML case the state would be the graph $G = (V, E)$ together with the labels/features and the sensors would be the human investigators. The set of possible sensors (actions) A would be every possible subset of the nodes V of our graph of a fixed size (where the fixed size indicates the amount of human investigators available). Both the set of possible states and sensors/actions would probably be huge.

There is a subtle difference between our problem domain and the domain of controlled sensing. In the controlled sensing domain, it is assumed that there are multiple types of sensors (of higher or lower fidelity). In the AML problem this is not the case: there are a limited amount of sensors and they are all of the same type. Because there is only a single type of sensor, the sensor usage cost $c(s_t, a_t)$ would just be a constant. However, as noted in the assumptions, in the real situation we only supply suggestions to the human investigators. They themselves decide what accounts they want to investigate. This decision making by the human investigators could perhaps be viewed as a low level sensor: it senses whether the humans find the node suspicious or not. If they do find it suspicious, the high level sensor gets deployed automatically.

One more thing is that it is not immediately clear how the sensor performance loss $d(s_t, b_t, a_t)$ would have to be defined in the AML case. In most other examples of controlled sensing, the sensor performance loss models the fact that ideally we would like to know exactly what state s_t we are in. In AML, this is not necessary.

As mentioned before, the state s_t is the entire graph G together with all the labels/features. This graph however, is huge and we are not interested in the graph in its entirety. We are only interested in the parts of the graph that indicate fraudulent nodes which, as mentioned before, can be considered anomalies and are therefore relatively sparse. So, instead of optimizing for observing the entire graph we would somehow like to optimize on observing only those parts of the graph that are of interest to detecting fraudulent behaviour. In fact, observing the graph is not the end goal of AML. The end goal is to find fraudulent behaviour. Observing the graph is only a way of achieving this.

So, the problem domain of AML does not necessarily directly translate into the problem domain of controlled sensing.

B.2 Optimal Search

The problem of optimal search is the problem of finding an object of interest, the target, within a search space [25]. Typically, the search space is defined as Euclidean n-space or a (possibly infinite) collection of cells. The target is than located at a point in this Euclidean n-space or inside one of the cells. These two types denote the continues and discrete search spaces.

It is assumed that the searcher has to its disposal a probability distribution for the location of the target at time zero. Moreover, there is a detection function d that models the probability of finding the target at a location given the time spent searching there and whether or not the target is actually at that location. Finally, there is some constraint on the effort the searcher is allowed to apply towards finding the target. This can apply to both stationary and moving targets.

One objective could be to find the optimal allocation of effort across the search space that maximizes the probability of finding the target. Another could be to optimize detection probability at each time instant or to minimize the mean time to find the target.

Example of an optimal search problem

One simple example of an optimal search problem would be the scenario of a search space consisting of two cells. The prior belief is that the target has a probability of $\frac{1}{3}$ of being in cell 1 and a probability of $\frac{2}{3}$ of being in cell 2. You can think of this problem as the problem of finding a valuable item in one of two drawers.

The detection function d could be of the form:

$$d(t) = 1 - e^{-t} \tag{B.3}$$

where t is the time spent searching in a drawer and $d(t)$ is the probability of finding the target in that drawer after searching for t hours given that it is in that drawer. This detection function is of this form because there typically is a non-zero probability of finding the target when first looking in the drawer (the searcher might be staring at the valuable item without recognizing it). Moreover, there is the effect of diminishing rate of returns: after hours spent searching for the item in the drawer, searching for a little longer will not drastically change the detection probability.

Suppose there is only a fixed amount of time T that the searcher can look for the item. The question to answer would now be: How to divide the search efforts between the two drawers to maximize the probability of detecting the item with time T?

Similarity to our problem

In the case of AML, we could consider a discrete action space where every node of our network is a cell. The target could be the fraudulent behaviour. Of course, in our case this means that there isn't a single target but rather an unknown amount of targets.

The target in AML would be a stationary target and the whole process could be modeled as a POMDP to incorporate the previous search history into a current belief of where the target is.

The detection function d in the case of AML would model the process of investigation by our human investigators. However, this function would need to have a non-trivial shape in order to model the complex human way of investigating.

One interesting difference between our AML model and the optimal search model is that in the optimal search model there are fixed size time steps. After every time step the decision has to be made to continue searching in the current cell or to preemptively stop the search in favor of searching in a different cell. In the AML problem the human investigators will decide when to stop 'searching' themselves. They will stop whenever they have accumulated enough evidence for or against fraud. The time it takes to reach this point will depend on whether the 'target' is at that 'cell' or not. In other words, it takes more time to collect evidence on a fraudulent node than it takes to determine whether a node is most likely not fraudulent. Moreover, there is no option of preempting the investigators since they decide what to do autonomously.

B.3 Active Learning

Active learning is a special case of machine learning in which a learning algorithm is able to interactively query the user (or some other information source), often

called the oracle, to obtain the desired outputs at new data points [7].

The problem of active learning can be formulated as follows. Suppose D denotes the total set of data under consideration. In each iteration the total set of data D can be divided into three categories:

- D_l: the labeled data points
- D_u: the unlabeled data points
- D_c: the chosen data points for which labels are requested from the oracle

The active learning problem is the problem of deciding which (previously unlabeled) set of points D_c to send to the oracle for labeling. Often, the problem starts with very few or no labeled data points and the objective is to reach a high classification accuracy with as few oracle queries as possible.

Sampling Scenarios

There are different scenarios in which active learning can be applied. The following are the three main types of scenarios [7]:

- **Pool-based sampling**: In this scenario, there exists a large pool of unlabeled data. Instances are drawn from that pool according to how informative these instances are expected to be.
- **Stream-based sampling**: In this scenario, instances are sampled in a stream. For each instance, it has to be decided whether to query the label or discard it.
- **Membership Query Synthesis**: In this scenario, the learner is allowed to draw any instances from the input distribution, notably including (and typically assumed to be) instances it generates itself.

The pool-based sampling scenario is most similar to the scenario of our problem. How to decide what samples to query is called the *Query Strategy*.

Query Strategies

There are a number of proposed query strategies in the literature that can be considered heuristics for the *informativeness* of an instance [7]:

- **Uncertainty Sampling**: The most informative instance is the one the learner is least certain about.

- **Query-By-Committee**: A committee of learners is maintained. The most informative instance is the one the committee disagrees on the most.
- **Expected Model Change**: The most informative instance is the one that leads to the greatest expected change of the current model.

Other query strategies exist that directly or indirectly try to minimize classification error such as Estimated Error Reduction, Variance Reduction or Fisher Information Ratio [7] (not a direct reference yet).

Similarity to our problem

On first impression, the domain of active learning shares a lot of properties with our problem domain. Firstly, the process of iteratively choosing a small subset of the data points to be labeled by some oracle is the same in both domains. Secondly, both domains share the principle research question: "Which instances to query and when?".

However, there are some subtle differences between the objectives of the problem domains. In active learning, the objective is to end up with a model that minimizes the classification error on all the data. However, in our problem domain, the objective function isn't necessarily the same. In our domain, the objective is to let the oracle label as many instances of a particular class (the fraudulent class) as possible. This subtle difference becomes clear in the case the learner has to decide between two instances to label:

1. An instance where the learner is very certain it is fraudulent, but this instance is not very informative for the classification.
2. An instance that is highly informative and will most likely improve classification, but is most likely not fraudulent.

In the active learning case the second instance would always be preferred over the first. But in our problem domain, the first instance might be more preferable. This can be thought of as an exploration versus exploitation issue.

Another, perhaps related, difference is that in a typical active learning environment, it is assumed the oracle always gives the true label with certainty and independently of any other queries or their order. However, in our problem domain, this assumption doesn't necessarily hold. In the case of AML, the human investigators may not return the true label and the labeling could potentially depend on the instances already labeled/available information. So, using a query strategy that simply optimizes for classification would neglect the sequential decision making aspect of our problem.

Appendix C

POMDP Solution Approaches

C.1 Exact Solutions

Ideally, if the full POMDP model is known, one would like to optimally solve the problem using an exact planning algorithm.

Most exact planning algorithms solve the POMDP by finding the optimal value function for all possible belief states. One such exact algorithm is the Value Iteration algorithm [26]. The value iteration algorithm makes clever use of the Bellman optimality equation for POMDPs:

$$V^*(b) = \max_{a \in A} \left[\sum_{s \in S} R(s,a)b(s) + \gamma \sum_{o \in O} p(o|b,a)V^*(b^{ao}) \right] \tag{C.1}$$

where b^{ao} is defined as the updated belief after taking action a and observing o:

$$b^{ao}(s') = \frac{O(o|s',a)}{p(o|b,a)} \sum_{s \in S} T(s'|s,a)b(s) \tag{C.2}$$

and $p(o|b,a)$ is the probability of observing o given you have belief b and take action a:

$$p(o|b,a) = \sum_{s' \in S} O(o|s',a) \sum_{s \in S} T(s'|s,a)b(s) \tag{C.3}$$

O and T denote the observation and transition probabilities respectively which are part of the POMDP model definition. $V^*(b)$ denotes the optimal value function V for belief b. The Bellman equation C.1 can be compactly written as:

$$V^* = H_{POMDP}V^* \tag{C.4}$$

where H_{POMDP} is called the Bellman backup operator.

For a POMDP with a state space of size n, the belief b is an n-dimensional vector. The value function V is therefore defined over an n-dimensional space. For the optimal value function V^*, equation (C.1) is satisfied for every belief point b. Computing the optimal value function V^* might seem intractable at

first. However, V^* admits to a particular structure: it is convex and piece-wise linear [27]. Its convexity can be intuitively understood by reasoning that at the corners of the belief simplex $\Delta(S)$ the value function will be high. This is because at the corners the state uncertainty is gone which allows for much better planning into the future.

C.1.1 Value Iteration

Value iteration algorithms can find the optimal value function for finite horizon POMDPs by successively applying the Bellman backup operator H_{POMDP}. The idea behind these algorithms is that the Bellman equation (C.1) defines the optimal value function for the current belief in terms of the optimal value function in terms of the belief in the next time step (after taking action a and observing o). Now, if there is only a single time step left to act, (C.1) reduces to:

$$V_1^*(b) = \max_a \left[\sum_s R(s,a)b(s) \right] \tag{C.5}$$

which is just a maximization over the weighted sum of the immediate rewards obtained in the next time step. In this case it is also easy to see that this value function is piece-wise linear. This is because the reward function $R(s,a)$ is only a function of state s and action a. In other words, it can be viewed as a set of $|A|$ vectors that are all $|S|$-dimensional: $\{\alpha_1^a(s)\}_a$. Every vector is associated with an action and defines a hyper-plane over the belief simplex. For every belief point b there is a single hyper-plane that has the highest value and the action associated with this hyper-plane is the optimal action to take in for that belief. The optimal value function can simply be written as an inner product of these vectors with the belief:

$$V_1^*(b) = \max_a \left[\sum_s R(s,a)b(s) \right] = \max_{\{\alpha_1^a\}_a} b \cdot \alpha_1^a \tag{C.6}$$

This structure of the optimal value function holds for any finite horizon $h > 0$ [28]. In other words, the optimal value function for any horizon h can be parameterized by a finite set of vectors or hyper-planes:

$$V_h^*(b) = \max_{\{\alpha_h^k\}_k} b \cdot \alpha_h^k \tag{C.7}$$

where $k = 1, \ldots, |V_h^*|$.

The trick behind value iteration lies in first defining the optimal value function for a single step horizon (C.5) and successively applying the Bellman backup operator to obtain the optimal value function for larger horizons.

The formulas above all hold for single belief vectors. If the belief space was finite and discrete, it would be possible to apply the formulas for every possible

state in every iteration. However, in POMDPs the belief space is defined over the unit simplex $\Pi(S)$ and is therefore continuous. So, applying the formulas for every possible belief is no longer possible.

However, this doesn't mean that finding the exact optimal value function is impossible. In general there are two approaches to doing this. One of those approaches focuses on enumerating all possible vectors of $H_{POMDP}V^*_{h-1}$, followed by pruning of dominated vectors.

C.1.2 Enumeration Algorithms

The idea behind Enumeration algorithms is to simply compute all the $\alpha-$vectors that are generated by applying H_{POMDP} to V^*_{h-1} given the known α-vectors in V^*_{h-1}. The issue is that there have to be $|A||V^*_{h-1}|^{|O|}$ vectors generated at each step. Many of these vectors could be dominated by others and therefore not part of V^*_h. Therefore, all enumeration algorithms perform some form of pruning to get rid of such dominated vectors before the next step. Differences can be in either pruning after generating all $|A||V^*_{h-1}|^{|O|}$ vectors, or pruning in an incremental fashion during the creation of the vectors.

How many vectors can be pruned is dependent on the problem. In the worst case none of the vectors can be pruned. In any case, enumerating these vectors takes a long time even for some small problems.

C.1.3 Witness Point Algorithms

The other approach to finding the exact optimal value function focuses on the fact that it is not necessary to perform the Bellman backup procedure for every possible belief. For every belief there exists a single optimal action that is associated with that belief. This action defines a hyper-plane over the belief simplex. We also know that the optimal value function consists of a finite set of such hyper-planes (many beliefs will share the same optimal action/hyper-plane). In theory, we would only have to perform the Bellman backup described in the section above over the finite set of beliefs (called witness points) that generate this optimal value function. The problem is, we don't necessarily know what the set of witness points is.

One algorithm called Cheng's Linear Support algorithm tries to find the witness points in the following way. It first picks a random belief point b in the belief space and finds the dominating vector for this belief point. This vector is known to be the dominating vector at b (by construction) and the dominating vector for some region of beliefs around b (due to the convex and piece-wise linear nature of V^*).

Let's assume for now that this single vector entirely defines V^* (which means

this vector dominates for every belief). If this assumption were false (which it most likely is), the largest error would be made at the boundaries of the region where this vector dominates (which we assumed to be the entire belief space). So, we take these boundary points (let's denote them b_i) and find the vectors that actually dominate for those belief points. This gives us a new set of vectors which are known to dominate at b and b_i and some regions surrounding those points.

We now iteratively find the boundary points for the regions in which these vectors dominate and construct the vectors that actually dominate on those boundary points. This is repeated until at some point no new vectors are found. This is the point that all dominating vectors are found and V^* is constructed in its entirety.

C.2 Approximate Solutions

Most algorithms that solve the POMDP exactly try to optimize the value function over the entire belief space. Those algorithms typically run into two problems: curse of dimensionality and curse of history [8]. The former is due to the fact that the dimensionality of the problem is equal to the number of states. In other words, a POMDP with n states has to be solved over an n-dimensional belief space. The latter refers to the fact that the number of belief-contingent plans increases exponentially with the planning horizon.

Both of these attributes contribute to the fact that, in general, solving POMDPs exactly is extremely difficult. In fact, finite-horizon POMDPs are PSPACE-complete [10] and infinite horizon POMDPs are even undecidable [11]. As a result, even small POMDPs are often computationally intractable. Therefore, there is the need for approximate solutions.

C.2.1 Point-Based Algorithms

One class of such approximate techniques is the class of Point-Based Value Iteration (PBVI) techniques [8]. Exact solutions optimize the value function over the entire belief space, PBVI techniques try to only optimize over parts of the belief space that are most informative. They do this by performing the point-based Bellman updates (as described in C.1.1) for a set of belief points B. The different PBVI methods differ in their way of selecting this set of belief points. These approaches often exhibit good performance with few belief points (relative to the size of the belief space), which addresses the curse of dimensionality.

The PBVI methods chose their set of belief points in different ways ranging from fast and naive ways to more sophisticated techniques. An example of a fast and naive way is to simply randomly sample your belief points from the

belief space. More sophisticated techniques only sample from the set of *reachable* beliefs. That is, beliefs that are actually reachable from the current prior belief (no point in focusing on beliefs that will never be encountered). This can either be done by building a tree of reachable beliefs starting from the current prior belief, or by sampling belief states by simulating random interactions in the POMDP environment.

Another approach is provided by PERSEUS [29], which aims to increase (or at least not decrease) the value function over the belief points B at each stage. The key idea is to only perform the Bellman backup for a (randomized) subset $\tilde{B}$ of B. New points are added to $\tilde{B}$ until the new V_{n+1} (resulting from only backing up points in $\tilde{B}$) is larger than or equal to V_n for every point in B.

A general property of point-based algorithms is that a trade off can be made between problem complexity and solution accuracy by increasing or decreasing the size of B.

C.2.2 MDP Based Heuristic Algorithms

Another approach is to use the value function of the MDP underlying the POMDP. Calculating the optimal value function of the underlying MDP is much easier compared to the POMDP because it does not have the exponential growth in the size of the observation space (solving an MDP is P-complete versus PSPACE-complete for POMDPs [10]).

One straightforward approach is to use the optimal value function for the underlying MDP (denoted $V^*_{MDP}(s)$). The idea is to assume we are in the state that is most likely according to our belief $b(s)$. So, we approximate the POMDP value function $\tilde{V}$ by

$$\tilde{V}_{MLS}(b) = V^*_{MDP}(\arg\max_s b(s)) \tag{C.8}$$

Another approach is to not use the most likely state, but instead take a weighted average over the states weighed by the belief:

$$\tilde{V}_{MDP}(b) = \sum_{s \in S} V^*_{MDP}(s) b(s) \tag{C.9}$$

A slightly more sophisticated variation of the above approach is the so called QMDP approximation. The key idea behind this method is to assume all partial observability disappears after a single step. The QMDP approximation defines the following Q-function:

$$Q^{MDP}_{t+1}(s,a) = R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') V^*_{MDP,t}(s') \tag{C.10}$$

This Q-function defines an α-vector (hyper-plane in the belief space) for each action a. So the POMDP value function is approximated by:

$$\tilde{V}_{QMDP}(b) = \max_{\{\alpha^a\}_a} b \cdot \alpha^a = \max_{\{Q^{MDP}(a)\}_a} b \cdot Q^{MDP}(a) \tag{C.11}$$

Note that the key idea behind this method is to assume all partial observability disappears after the next step. This means that this method will never consider policies that require repeated information gathering steps.

The above methods require you to have a prior belief, and perform a belief update after every step, based on the action taken and observation received.

C.3 Offline vs Online Approaches

There is a distinction to be made between POMDP solving approaches: offline vs online. Offline and online algorithms are typically used in different settings. The offline setting is one in which all the planning/learning is done before any execution takes place. That is to say, an offline algorithm has to derive the optimal policy for every possible belief state before execution. The online setting is one in which planning/learning and execution are interleaved. Typically, in the online setting the algorithm tries to derive the optimal action to be taken in the current belief state. Once found, this action is executed and the new belief state is observed. The algorithm then tries to find the optimal action in the new current state etc. There are some typical advantages and disadvantages between offline and online methods that are due to the difference in setting.

As mentioned before, offline methods need to derive the optimal policy for every possible belief state before execution. For many problems this takes too large a time to solve. Moreover, they are typically not very robust against changes in the dynamics of the environment [12]. This is because, if during execution the dynamics of the environment change, the policy computed offline using the previous dynamics might not be as relevant anymore. Similarly, going from a low fidelity simulation to the real world could potentially also be thought of as a change in dynamics of the environment (from simulation to real). The limiting factor of online approaches is typically that it needs to run in real-time. In other words, computation time is limited to the amount of time there is between actions.

As opposed to offline methods, online methods don't have to find the optimal policy for every possible belief. They just have to find the optimal action for the current belief. This is very useful if the size of the belief space becomes very large (curse of dimensionality). Only finding the optimal policy for the current belief/state, and repeating this after every time step also makes online approaches more robust against changes in the dynamics of the environment. The same holds for uncertainty on the dynamics of the environment, a mistake

in this is not propagated through more than a single action into the future. A limiting factor of online approaches is typically that it needs to run in real-time. In other words, computation time is limited to the amount of time there is between actions. To deal with the real-time constraint, many online approaches continually keep track of the best found solution so far. This means that they can be terminated at any time and still supply the best solution so far.

A combination of both approaches is also possible. A lot of online approaches can be used as a way to improve upon some baseline (offline) solution.

C.4 Online Solutions

The general structure of online approaches is that they build a tree of the search process. The root of the tree is always the current belief state. From the root, taking an action will lead to an observation which together with the taken action defines a new belief (using a belief update). In this way the search tree is built by looking at several sequences of actions and observations up to a certain depth. The value of a belief state is propagated from the leaf nodes all the way to the root using Bellman's equation.

The search tree is built by first initializing it to contain only the root node (current belief). In each step the tree is expanded by selecting a leaf node from which it should pursue the search. This leaf node is expanded, and the values of the added nodes are usually estimated using some approximation computed offline. The tree is iteratively build in this way until a termination condition is met (typically a search time limitation). After this the action (edge outgoing from the root node) with the best value found so far is chosen to be executed.

Most of the online algorithms differ in what reachable beliefs to explore (how they construct the search tree). They can be classified into three catergories: Branch-and-Bound Pruning, Monte Carlo Sampling and Heuristic Search.

C.4.1 Branch-and-Bound Pruning

The aim of this technique is to prune nodes that are known to be suboptimal in the search tree. They manage this by storing upper and lower bounds on the value function at each node of the tree. If a particular action in a belief state has an upper bound that is lower than the lower bound of a different action, this action can be pruned.

One algorithm that employs this technique is called Real Time Belief Space Search (RTBSS) [30] [31]. This algorithm, in order to maximize pruning, expands the actions in descending order of their upper bound (so highest upper bound first). The reasoning behind this, is that if an action a has the highest upper bound, it cannot be pruned by first expanding a different action. This is because

the lower bound of those other actions can never exceed the upper bound of action a (since they cannot exceed their own upper bounds, which by definition are lower than the upper bound of a). So this method avoids the scenario where an action is expanded that could have been pruned if the actions were expanded in a different order. Another advantage to expanding in this descending order, is that once an action is found that can be pruned, it is immediately possible to prune all the other unexpanded actions. This is because their upper bounds are necessarily lower than the one of the action that was just pruned. The algorithm performs a depth first search (up to a predetermined expansion depth). This means it can prune actions faster, since depth first search leads to tighter bounds.

The efficiency of these methods depend largely on the precision of the upper and lower bounds computed offline. One drawback is that it explores all observations equally. This means that if the number of observations is large, the depth of the tree has to be very limited. However, this does mean it can guarantee to improve the original (offline) solution by a factor γ^D (where D is the expansion depth).

C.4.2 Monte Carlo Sampling

Monte Carlo sampling approaches solve the problem of small expansion depth by expanding the search tree on a sampled subset of observations. One such approach is the McAllester and Singh algorithm [32] which samples from a generative model. Sampling in this way relieves the complexity of the algorithm in exchange for a less precise estimate. However, it assumes that a few samples can be good enough because the observations contributing most to the value of an action are those with the highest probability of occurring (and therefore the highest probability of being sampled).

Another Monte Carlo sampling approach is the rollout algorithm [33]. This algorithm uses an initial rollout policy to estimate the value of every action at the root node. It then simply chooses the action with the highest estimated value. An improvement on this method is where instead of a single initial policy a set of initial policies is tested. For every action and every policy in the initial set a number of samples are rolled out and the sample average is used to determine the value of the action. This method is guaranteed to perform at least as well as the initial policy with enough samples. However, it does not expand the search tree beyond the first actions. This means that if the initial policy cannot be improved by changing just a single action, this method will never improve upon it.

One drawback of Monte Carlo sampling approaches is that it does not guarantee correct propagation of the lower and upper bound. This means that pruning is not possible. These approaches differ from Branch-and-Bound approaches in that they expand a node by evaluating each action (through Monte Carlo sampling of the observations), rather than evaluating the actions in a particular order

and pruning actions whenever possible. This can be viewed as trading off the expansion factor due to the observation space versus the expansion factor due to the action space. As a result, Monte Carlo sampling approaches can be difficult if the number of actions $|A|$ is large.

C.4.3 Heuristic Search

The previous two methods tried to reduce the branching factor of the search. Heuristic search however, tries to use heuristics to pick the most relevant fringe nodes. The most relevant ones are the ones that allow the algorithm to make good decisions as fast as possible.

C.5 Model-based vs Model-free Approaches

Most of the methods discussed so far require explicit knowledge of the POMDP model. Specifically, in order to perform a belief update or Bellman backup you require explicit knowledge of the transition probabilities $T(s'|s,a)$, reward function $R(s,a)$ and/or observation probabilities $O(o|s',a)$. These functions are defined over S, A and O. However, these spaces can be too large to store those functions explicitly. For instance, one way to store the transition probabilities $T(s'|s,a)$ would be to have a 3-dimensional matrix over $SxAxS$. However, as soon as the state space gets to a certain size, storing this matrix becomes impossible. This is an issue all *model-based* (because they explicitly depend on the model) methods share. Model-based approaches become impractical for large or complex problems.

For larger problems we require so called *model-free* approaches. They are called model-free because they do not require explicit knowledge of the POMDP model. They typically only require implicit knowledge of the model through interaction with a generative model $\mathcal{G}$. A generative model $\mathcal{G}$ is function which takes as input a state s_t and action a_t and generates as output the next state s_{t+1}, observation o_{t+1} and reward r_{t+1}

$$(s_{t+1}, o_{t+1}, r_{t+1}) \sim \mathcal{G}(s_t, a_t) \tag{C.12}$$

Generative models allow on-the-fly samples from the distributions $T(s'|s,a)$, $R(s,a)$ and $O(o|s',a)$ at low computational cost. An example of such an approach is the Monte Carlo Sampling approach discussed in section C.4.2.

Appendix D

Large Observation Space

The POMCP algorithm uses Monte-Carlo sampling of states for both the tree search and belief state updates to deal with large state spaces. Consequently, it outperformed all the previous POMDP solvers on large problems.

A large state space is not the only thing that complicates a POMDP however. Some real world problems have not just large state spaces, but also large observation spaces. There are several extensions to the POMCP algorithm that try to solve problems with large state *and* observation spaces.

POMCP builds the belief tree by sampling observations from a generative model. When at belief node h and taking action a, every new observation o received results in a new history, resulting in a new belief node hao. The issue with large observation spaces is that the probability of obtaining the same observation in two separate simulations becomes smaller and smaller as the size of the observation space grows. In other words, if we are in h and perform a, the probability of sampling observation o in two separate simulation runs would be negligible if the observation space is large enough.

So, because each simulation will most likely sample a new observation, it will most likely create a new belief node. In this scenario our search tree would not extend deeper than the first layer (see figure D.1 from [18]). Remember that after every creation of a new belief node, the rollout policy is applied to a state sampled from the belief. Solutions of this algorithm will closely resemble QMDP solutions. Recall that a QMDP solution assumes all partial observability disappears after the next step and will therefore never consider repeated information gathering steps.

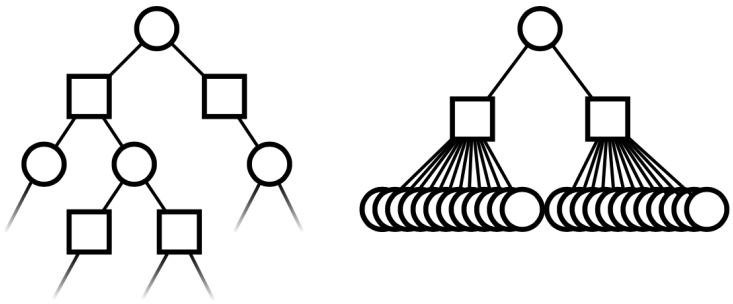


Figure D.1: Example showing that for large observation spaces the search tree never extends beyond the first layer.

Furthermore, if we want the POMCP algorithm to improve upon some baseline solution (perhaps found offline), this can be achieved by using this baseline as the rollout policy. The issue than becomes that, because the tree will never extend deeper than the first action, this method will never discover improvements on the baseline that differ from it in more than one action.

D.1 Continuous Spaces

The most extreme example of a large action space is the continuous case. The issue with a continuous space is that the probability of a 'hit' (second occurrence of an action) is zero. The authors in [18] describe two algorithms that deal with POMDPs with continuous state, action and observation spaces.

The POMCP algorithm can already handle continuous state spaces due to its Monte-Carlo sampling of states. However, as detailed above, it runs into problems when the action and/or observation spaces are large (let alone infinite). Their algorithms are based around the idea of Double Progressive Widening (DPW).

The name progressive widening comes from the fact that they hard limit the amount of children a node can have by $k * n^{\alpha}$. If the limit of children is reached, instead of creating a new child they sample from the existing ones with probability proportional to how often a child occurred. This limit term increases with the amount of times n the node is visited. In this way the tree progressively widens. It is called double progressive widening because this concept is applied to both the observation space and the action space.

D.1.1 POMCP-DPW

The first solution they describe is the POMCP-DPW solution. This is simply POMCP but with double progressive widening for the action and observation spaces. This solution may converge to the optimal solutions for POMDPs with large but discrete observation spaces. However, [18] prove that for a continuous observation space, this solution converges to the QMDP solution. This can be understood intuitively because every node other than the root only has a single state representing the belief (since the probability of a 'hit' is zero). So, from those nodes onwards the rollout just solves the MDP problem, not the POMDP. Regular POMCP and DESPOT both share this same characteristic.

If the POMCP-DPW algorithm reaches the observation limit, it would sample from the existing observations and then uniformly sample a state from the belief of that observation. The problem in continuous spaces is that an observation is ever only 'hit' once, meaning the belief of that observation is a particle filter with a single particle (state). To solve this issue of state particle dilutions, they propose the Partially Observable Monte-Carlo Planning with Observation Widening (POMCPOW) method.

D.1.2 POMCPOW

In the case the observation limit isn't reached yet, POMCPOW, just like POMCP-DPW, samples a new state s' and observation o from the generative model. Both algorithms then update the tree with the new observation o and add the new state s' to the particle filter of hao. One difference between POMCPOW and POMCP-DPW is that the latter uses an unweighted particle filter, whereas the former uses a weighted one (with weights proportional to $O(o|s', a)$). Furthermore, in the POMCP-DPW algorithm, the previously samples state s' is used for the continuation of the simulation. In POMCPOW on the other hand, a new state s'' is sampled from the belief at hao, proportional to their weights.

If the observation limit *is* reached, the POMCPOW algorithm still samples a new state s' and observation o from the generative model. However, because the observation limit is reached, o cannot be added to the tree (if o isn't already in there, which in the continuous case it will never be). Instead, an observation o' is sampled from the existing observations and s' is added to the particle filter of this node (with weight $O(o'|s', a)$).

The key result from those differences is that in the POMCPOW algorithm the particle filter of a belief node always has as many particles as amount of times the node was reached, avoiding the issue of particle dilution.

D.2 DESPOT

Another algorithm that tries to deal with large observation spaces is based on the DESPOT algorithm. DESPOT stands for Determinized Sparse Partially Observable Tree [34]. It is a variation on POMCP that aims to improve on POMCPs worst case performance. A standard belief tree of height D has $O(|A|^D|Z|^D)$ nodes. If the observation space $|Z|$ is large, this leads to a huge amount of nodes.

The DESPOT algorithm doesn't build a standard belief tree, instead it builds a Determinized Sparse Partially Observable Tree (DESPOT). The DESPOT is a belief tree that is build only from a set of K so called scenarios. These scenarios are randomly sampled a priori from the generative model. A DESPOT than consists of belief tree obtained by evaluating all possible action sequences for every single scenario. So it differs from a regular belief tree in that it only contains the observations from the K scenarios. So a DESPOT contains all the action branches but not all of the observation branches. The amount of nodes in a DESPOT of height D is $O(|A|^D K)$ (since every scenario covers all $|A|^D$ possible policies). So, for POMDPs with large observation spaces, the DESPOT contains significantly less nodes than a regular belief tree (see figure D.2 from [34]).

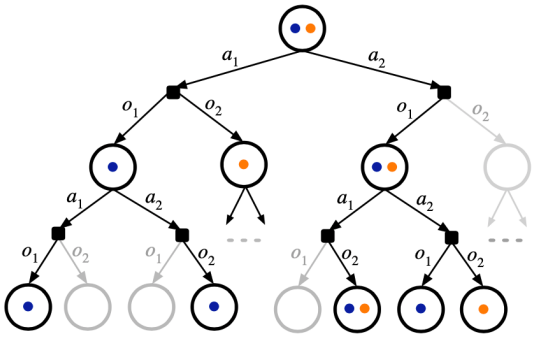


Figure D.2: Example showing the DESPOT contains less observation nodes than the regular belief tree.

D.2.1 Scenario

A scenario is defined through the use of a *deterministic generative model*. The deterministic generative model is defined to deterministicaly provide the POMDPs next state and observation given a state, action and real scalar ϕ:

$$(s_{t+1}, o_{t+1}) \sim g(s_t, a_t, \phi) \tag{D.1}$$

It is defined in such a way that if ϕ is uniformly distributed over $[0, 1]$, than g should be distributed according to $p(s_{t+1}, o_{t+1}) = T(s_{t+1}|s_t, a_t)O(o_{t+1}|s_{t+1}, a_t)$. With this deterministic generative model in mind, a scenario Φ is now defined as

a start state s_0 (sampled from the initial belief) and a sequence of real numbers ϕ_i uniformly sampled over $[0, 1]$:

$$\Phi = (s_0, \phi_1, \phi_2, ...) \tag{D.2}$$

Given the deterministic generative model g and an action sequence $\bar{a} = (a_0, a_1, a_2, ...)$, a scenario Φ now defines a trajectory $(s_0, s_1, a_1, o_1, s_2, a_2, o_2, ...)$. This trajectory traces out a history (descent path) of the belief tree: $(a_1, o_1, a_2, o_2, ...)$.

So, a DESPOT is created by sampling K such scenarios and evaluating them for all possible action sequences (all possible policies).

D.2.2 Basic Algorithmic Structure

The key idea behind a DESPOT algorithm is to approximate the optimal policy by searching through the space of *all* policies on a *subset* of scenarios. This is in contrast to for instance POMCP, which searches through a *subset* of policies (due to computational limitations) on the set of *all* scenarios. So in a typical DESPOT algorithm, you first create the full DESPOT ($O(|A|^D K)$ nodes), then find the best policy on this DESPOT.

There are various versions of the DESPOT algorithm. One is the Basic DESPOT (B-DESPOT). This algorithm simply performs normal tree search where the value of an action is simply the average of the values of that action over the different scenarios.

One issue with basic DESPOT is that it can overfit to the sampled scenarios. The Regularized DESPOT (R-DESPOT) tries to deal with this. The authors of [34] derive a lower bound on the actual value $V_\pi(b_0)$ of a policy π in terms of the estimate of this value $\hat{V}_\pi(b_0)$ obtained from a DESPOT. They rewrite this lower bound into a regularized utility function $\hat{V}_\pi(b_0) - \lambda|\pi|$ and show that maximizing this will lead to a near-optimal policy with high probability (if a small optimal policy π^* exists). The R-DESPOT algorithm finds the policy that maximizes this regularized utility function by performing bottom up dynamic programming on the DESPOT.

D.2.3 AR-DESPOT

The B-DESPOT and R-DESPOT algorithm have limitations to their online performance because they have to create the entire DESPOT every turn. The Anytime Regularized DESPOT (AR-DESPOT) deals with this by creating a version of R-DESPOT that can be terminated at anytime and still provide the best solution found so far.

The AR-DESPOT algorithm builds a partial DESPOT for as long as time permits, once time is up, it performs the R-DESPOT algorithm on the partial

DESPOT by maximizing the regularized utility. It builds the partial DESPOT using a combination of heuristic search and branch-and-bound pruning (using lower and upper bounds on the value).

It performs the heuristic search by running trials over the DESPOT that store upper and lower bounds on every belief node. A trial is performed by choosing actions that maximize the upper bound on the action node and choosing the observation nodes that maximize a so-called weighted excess uncertainty. This weighted excess uncertainty is a measure of how close the upper- lower-bound gap is compared to a desired one weighted by the amount of trials that have passed this node.

The initial upper bound can be set by taking the average of the upper bound over all scenarios passing through the particular node. The upper bound of any particular scenario is the maximum value achieved over any policy on that scenario. This can be computed in $O(K|S||A|D)$ time. To construct an initial lower bound, it is possible to use any policy on all the scenarios and take the average of the total discounted rewards (average over scenarios).

D.3 α-DESPOT

The DESPOT algorithm can handle larger observation spaces than POMCP due to it only searching over a finite set of scenarios. However, it still struggles with really large observation spaces for the same reason POMCP does: if the probability of obtaining the same observation twice are negligible, the K scenarios will result in K different observations, leading to belief nodes with single state particle filters, leading to policies over-optimistic policies that underestimate the uncertainties (see figure D.3 from [35]).

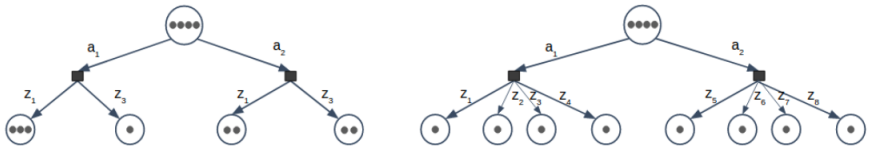


Figure D.3: Example showing that for large observation spaces the DESPOT never extends beyond the first layer.

The DESPOT with α-vector update (DESPOT-α) algorithm deals with this particle divergence by having so called sibling belief nodes share their particles [35]. Sibling belief nodes are belief nodes that have histories (descent paths) that differ only in the last observation received. In the DESPOT-α algorithm particles aren't partitioned into belief nodes based on the observations the particle generates. Instead, every child belief node inherits all its parents particles in a

weighted particle filter. The weight of the particle is determined by the likelihood of the particle generating that observation $O(o_{t+1}|s_{t+1}, a_t)$. Note that this does require explicit knowledge of the observation probabilities, but the weight update is only based on the states found in the particle filters and therefore can be performed efficiently for problems with large state spaces.

Another attribute of DESPOT-α is that it shares the value function calculation among sibling belief nodes.

DESPOT-α is designed to scale up to problems with extremely large observation spaces and can scale to complex real-world problems which require very fast decision making and have complex dynamics [35].

Appendix E

Extended Results

E.1 POMCP-UCD Implementation

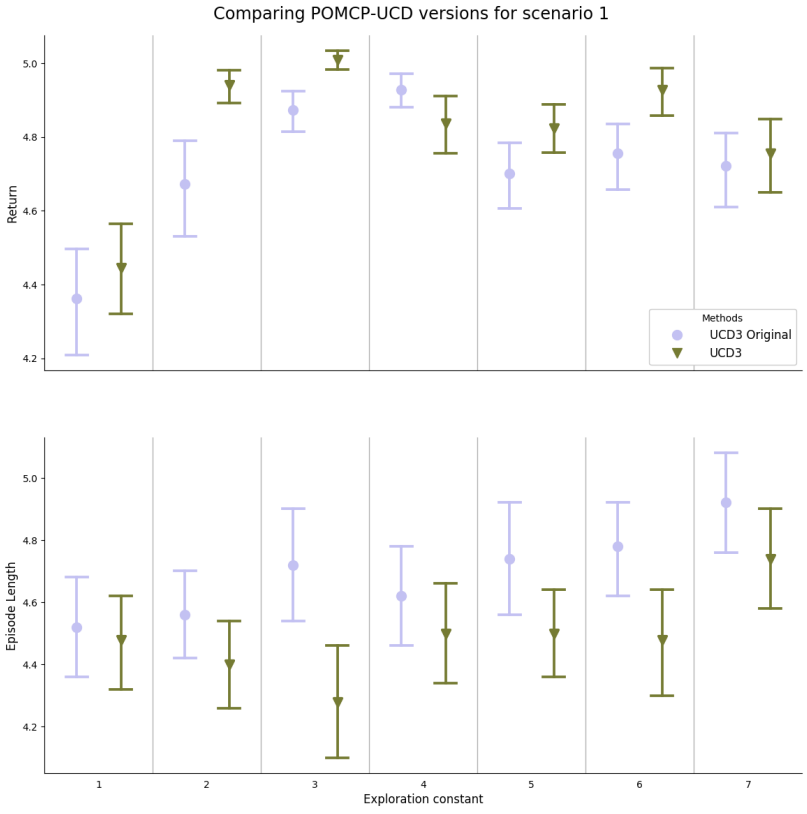


Figure E.1: Comparison between the two POMCP-UCD versions with depth parameter $d = 3$. Shown are the averages and 95% confidence intervals over 50 runs for exploration constants in the range $c = \{1, 2, 3, 4, 5, 6, 7\}$ and 1100 simulations per timestep.

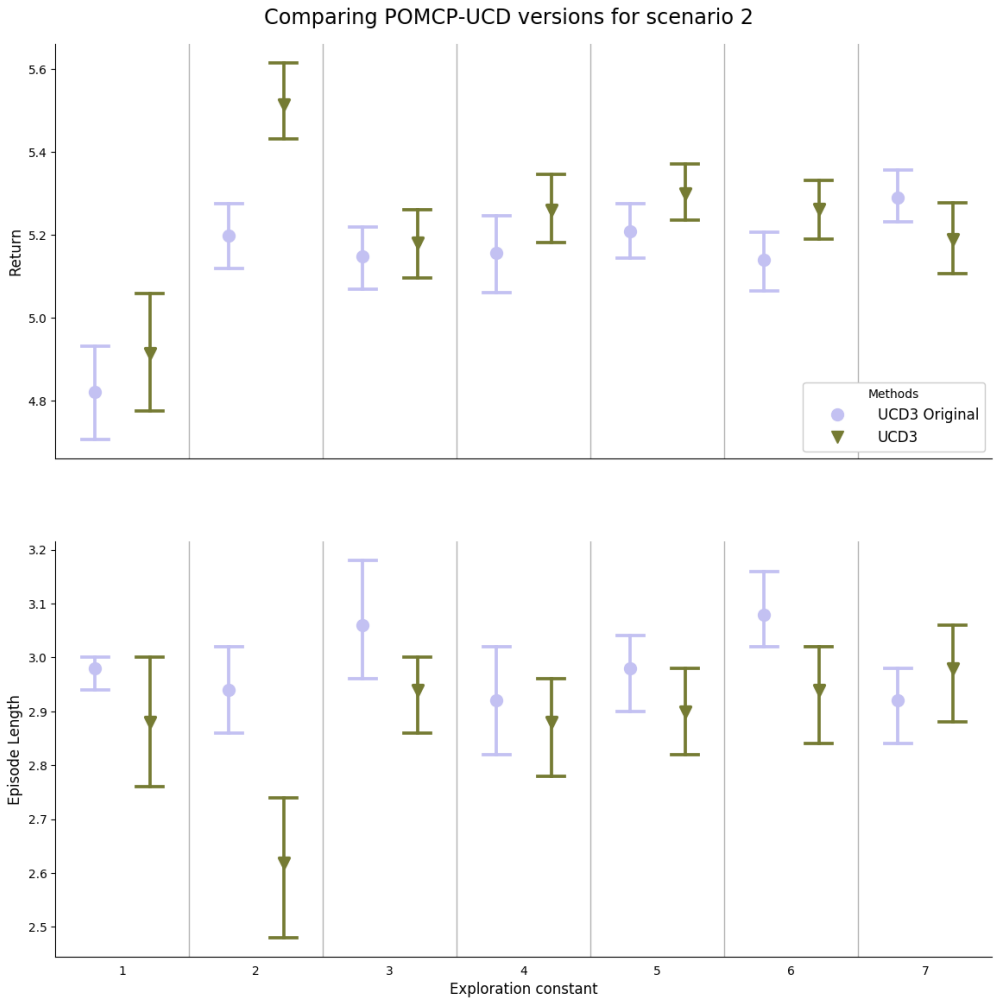


Figure E.2: Comparison between the two POMCP-UCD versions with depth parameter $d = 3$. Shown are the averages and 95% confidence intervals over 50 runs for exploration constants in the range $c = \{1, 2, 3, 4, 5, 6, 7\}$ and 1100 simulations per timestep.

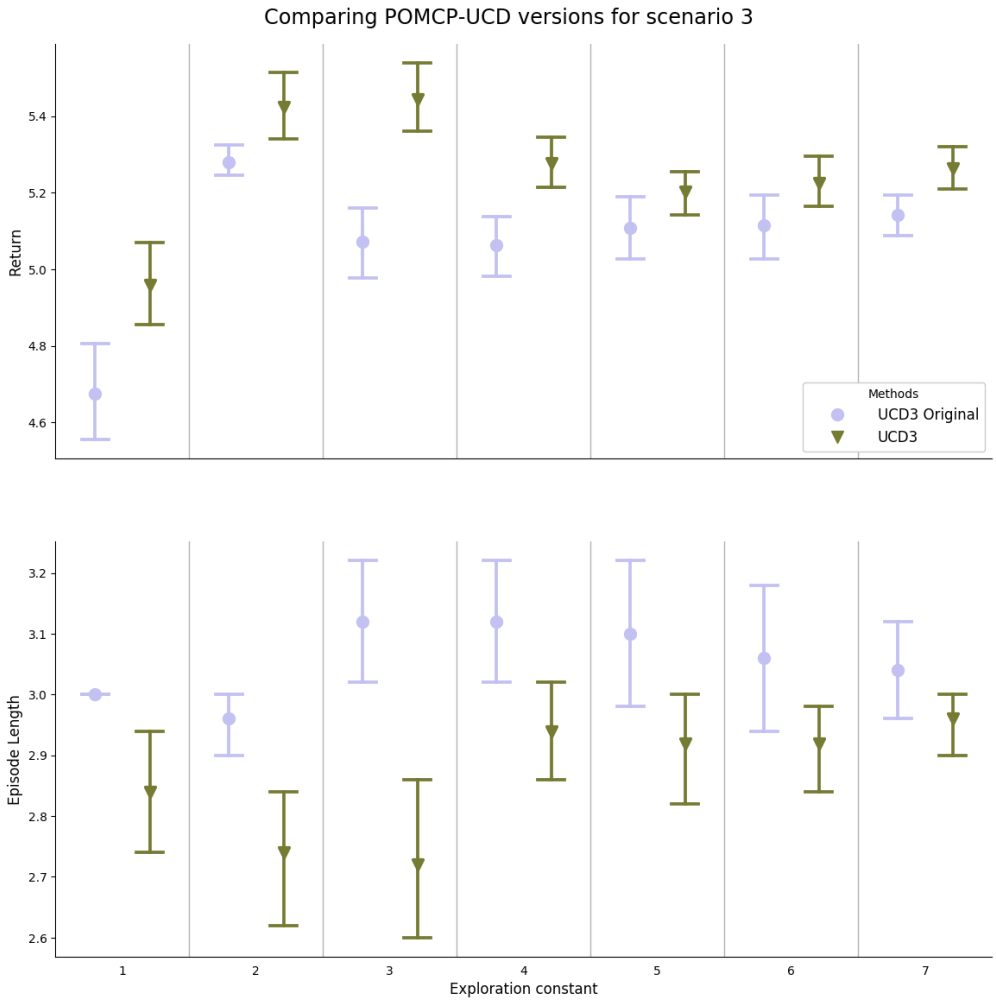


Figure E.3: Comparison between the two POMCP-UCD versions with depth parameter $d = 3$. Shown are the averages and 95% confidence intervals over 50 runs for exploration constants in the range $c = \{1, 2, 3, 4, 5, 6, 7\}$ and 1100 simulations per timestep.

E.2 POMCP-UCD Parametrizations

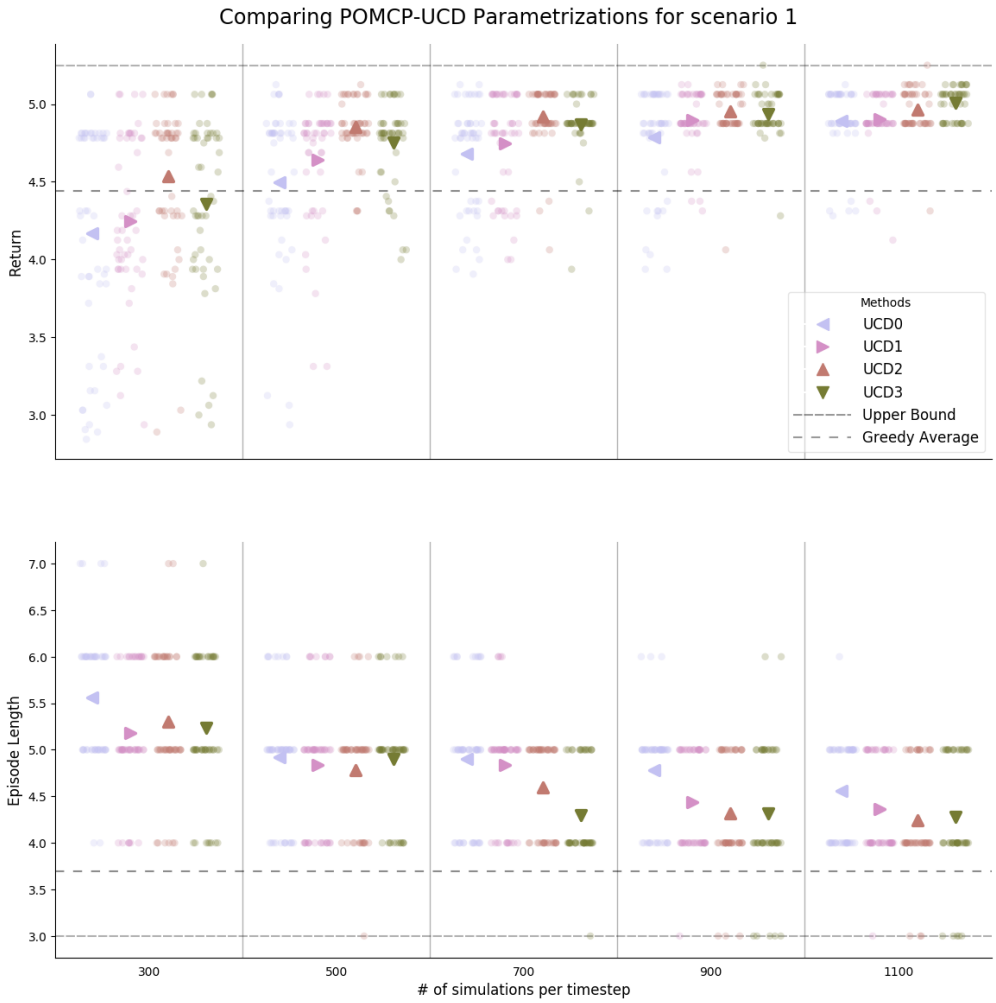


Figure E.4: The max over exploration constants $c = \{1, 2, 3, 4, 5, 6, 7\}$ for all possible depth parametrizations $d = \{0, 1, 2, 3\}$ of POMCP-UCD. Shown are 50 runs with simulations per time step in the range of $\{300, 500, 700, 900, 1100\}$. The markers denote the averages over those 50 runs.

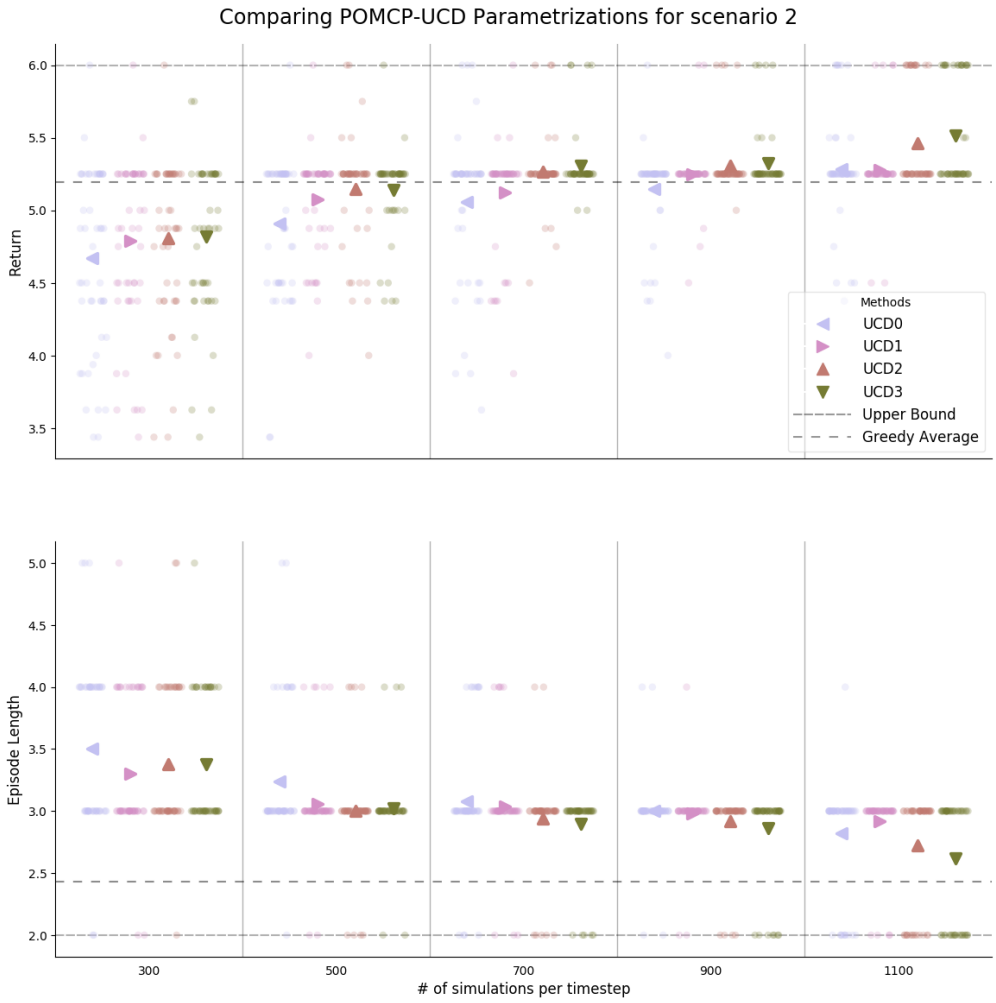


Figure E.5: The max over exploration constants $c = \{1, 2, 3, 4, 5, 6, 7\}$ for all possible depth parametrizations $d = \{0, 1, 2, 3\}$ of POMCP-UCD. Shown are 50 runs with simulations per time step in the range of $\{300, 500, 700, 900, 1100\}$. The markers denote the averages over those 50 runs.

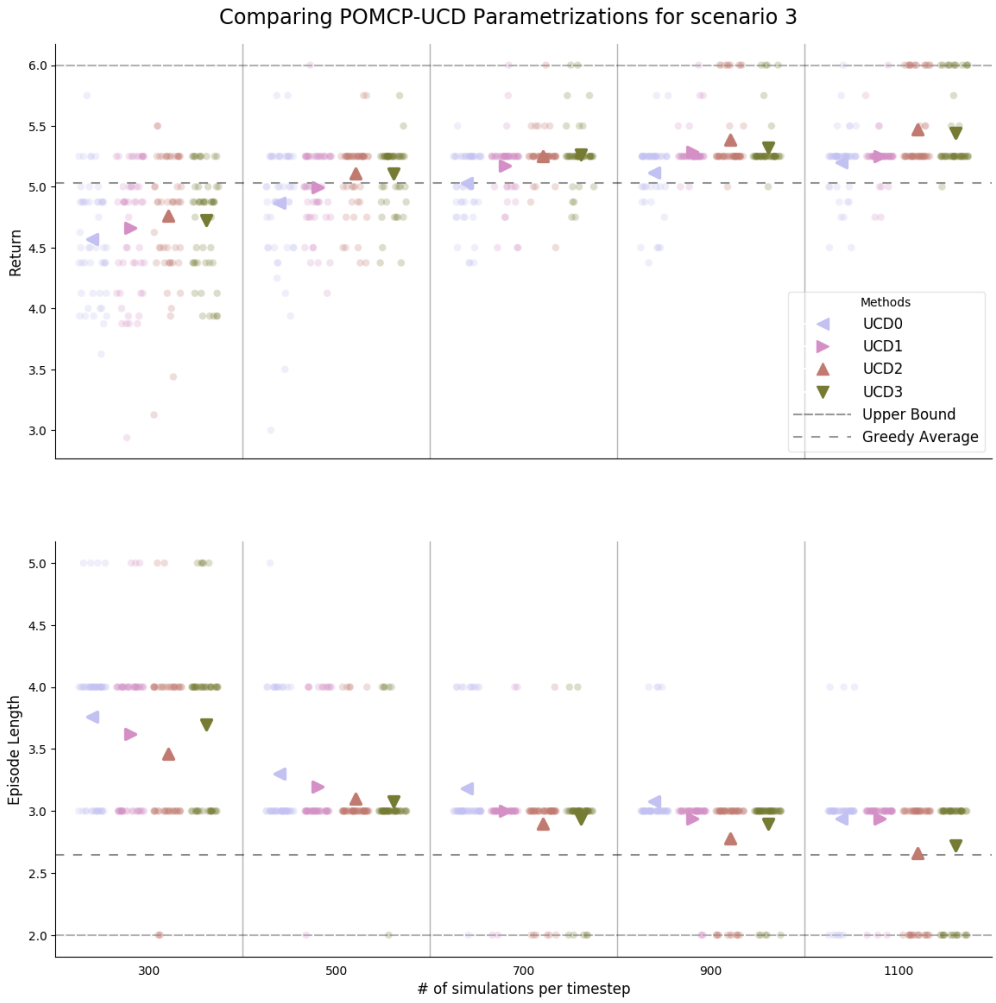


Figure E.6: The max over exploration constants $c = \{1, 2, 3, 4, 5, 6, 7\}$ for all possible depth parametrizations $d = \{0, 1, 2, 3\}$ of POMCP-UCD. Shown are 50 runs with simulations per time step in the range of $\{300, 500, 700, 900, 1100\}$. The markers denote the averages over those 50 runs.

E.3 All Methods

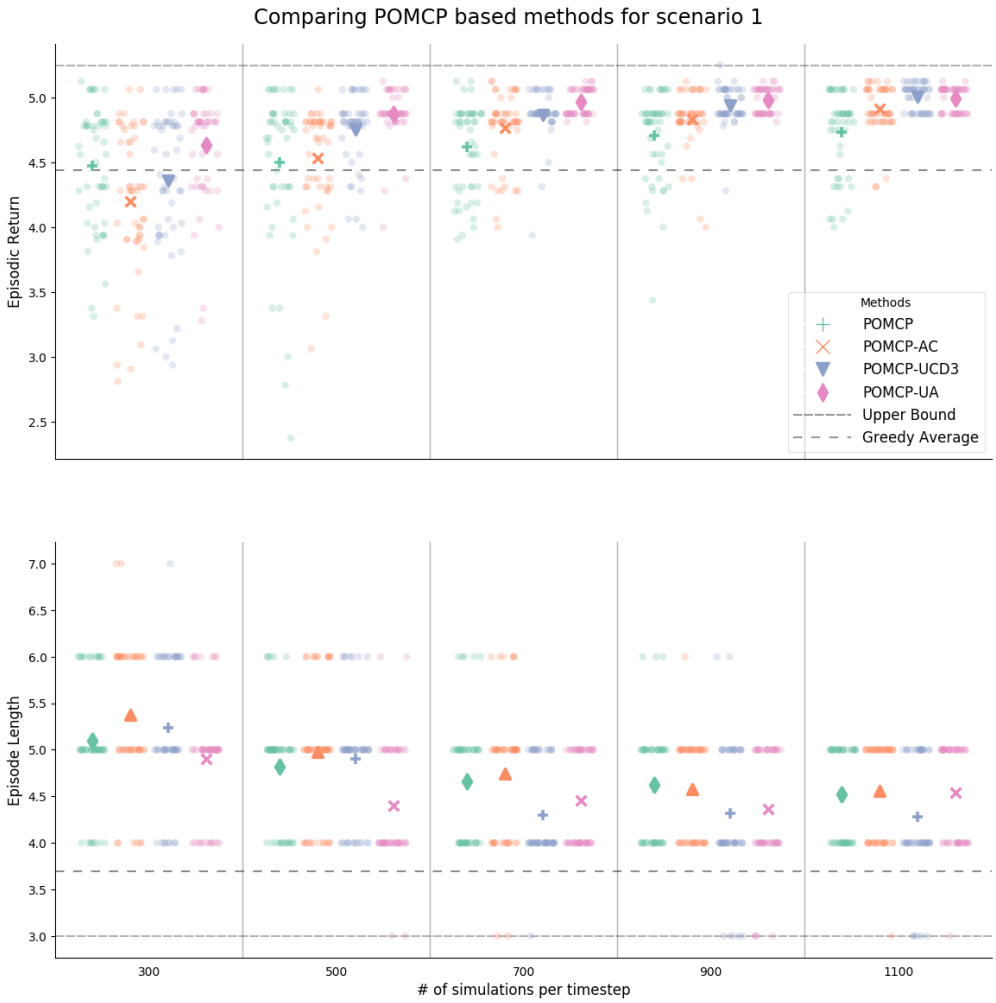


Figure E.7: The max over exploration constants $c = \{1, 2, 3, 4, 5, 6, 7\}$ for all POMCP based methods. Shown are 50 runs with simulations per time step in the range of $\{300, 500, 700, 900, 1100\}$. The markers denote the averages over those 50 runs.

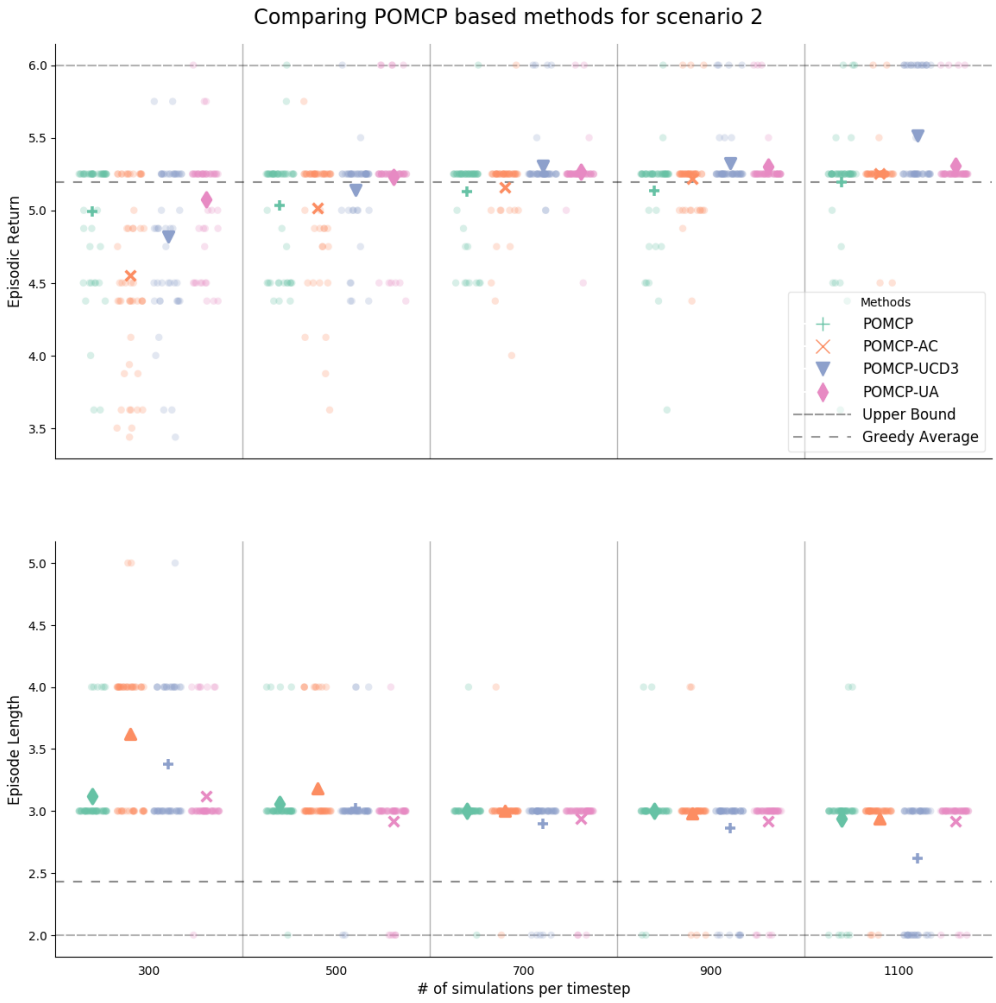


Figure E.8: The max over exploration constants $c = \{1, 2, 3, 4, 5, 6, 7\}$ for all POMCP based methods. Shown are 50 runs with simulations per time step in the range of $\{300, 500, 700, 900, 1100\}$. The markers denote the averages over those 50 runs.

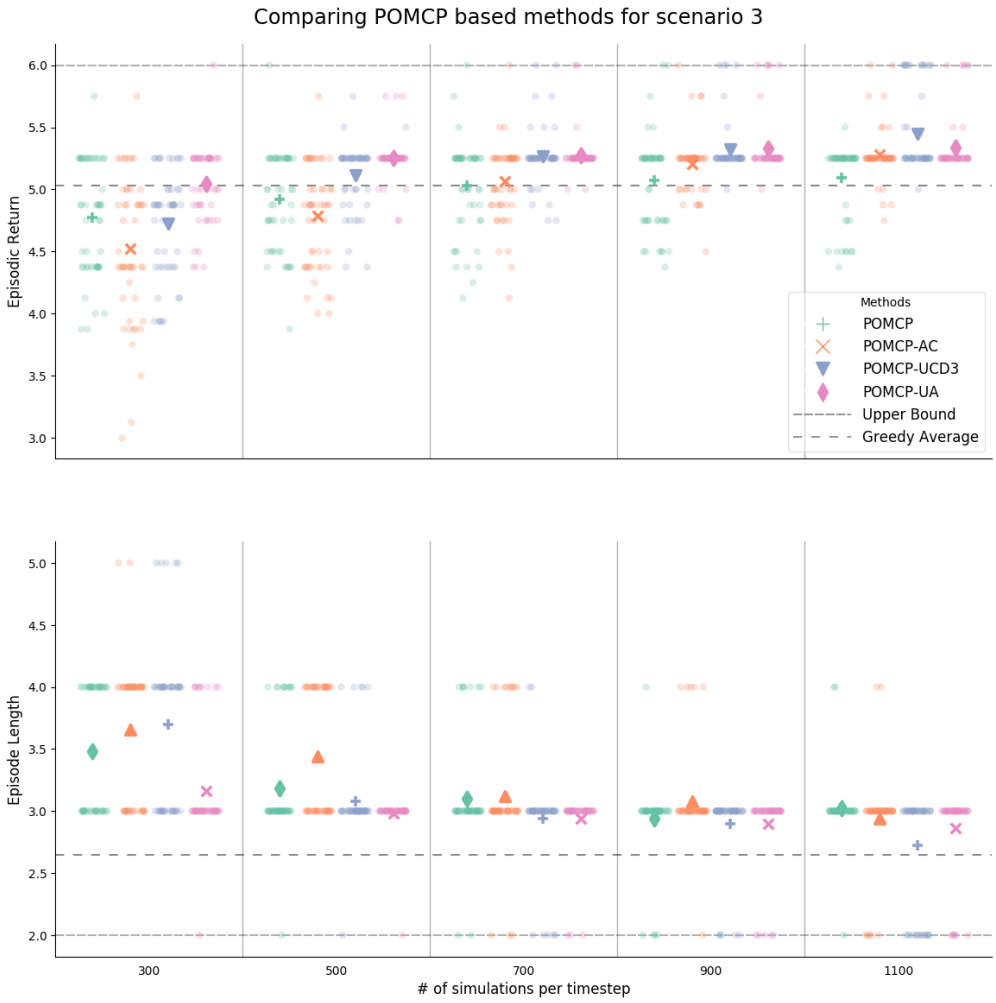


Figure E.9: The max over exploration constants $c = \{1, 2, 3, 4, 5, 6, 7\}$ for all POMCP based methods. Shown are 50 runs with simulations per time step in the range of $\{300, 500, 700, 900, 1100\}$. The markers denote the averages over those 50 runs.