

TRINITY COLLEGE DUBLIN

SCHOOL OF COMPUTER SCIENCE AND STATISTICS

M.SC IN COMPUTER SCIENCE

---

**Partitioning POMDPs for multiple input types, and  
their application to dialogue managers**

---

*Author:*  
Grace COWDEROY

*Supervisors:*  
Dr. Carl VOGEL  
Dr. John KELLEHER  
Prof. Nick CAMPBELL

Submitted to the University of Dublin, Trinity College,  
November 27, 2018



Coláiste na Tríonóide, Baile Átha Cliath  
Trinity College Dublin

Ollscoil Átha Cliath | The University of Dublin

# Partitioning POMDPs for multiple input types, and their application to dialogue managers

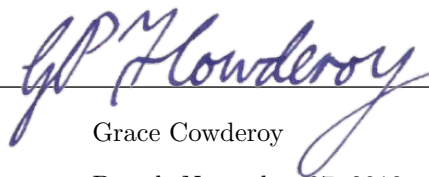
Approved by Dissertation Committee:

Saturnino Luz

Robert Ross

# Declaration

I declare that this report details entirely my own work. Due acknowledgements and references are given to the work of others where appropriate. I agree to deposit this thesis in the University's open access institutional repository or allow the Library to do so on my behalf, subject to Irish Copyright Legislation and Trinity College Library conditions of use and acknowledgement.



---

Grace Cowderoy

Dated: November 27, 2018

# Abstract

This research looks at the POMDP models used for dialogue management within Spoken Dialogue Systems (SDS). In particular, it examines the difficulty of handling multimodal inputs. It proposes a generalisation of the POMDP model in order to tackle this.

This model is shown, via a Car Advisory system, to improve tractability for multimodal inputs.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Spoken Dialogue Systems . . . . .	1
1.2	Aims . . . . .	2
1.3	Significance of Study . . . . .	2
1.4	Outline of Thesis . . . . .	3
<b>2</b>	<b>Current Literature</b>	<b>4</b>
2.1	Human-Computer Interactions . . . . .	4
2.1.1	Mutual Understanding . . . . .	6
2.2	Mathematics of POMDPs . . . . .	8
2.2.1	Policies . . . . .	10
2.3	Literature Review . . . . .	12
2.4	Current problems . . . . .	14
<b>3</b>	<b>Partitioning the POMDP</b>	<b>17</b>
3.1	Learning a POMDP . . . . .	17
3.2	Motivation for Partitioning . . . . .	18
3.3	Partitioning the POMDP state space . . . . .	20
3.3.1	Defining the model . . . . .	20
3.3.2	Update Function . . . . .	21
3.3.3	Dependent Partitions . . . . .	22
3.3.4	Theoretical Results . . . . .	23
<b>4</b>	<b>Evaluation</b>	<b>25</b>
4.1	Programming the model . . . . .	26
4.2	Comparison software . . . . .	26
4.3	Results . . . . .	27
4.4	Discussion . . . . .	28

<b>5 Conclusion</b>	<b>29</b>
5.1 Future Work . . . . .	30
5.2 Final Remarks . . . . .	31
<b>List of Figures</b>	<b>32</b>
Bibliography . . . . .	33
<b>Appendix</b>	<b>36</b>

# Chapter 1

## Introduction

This dissertation looks at the current methods and procedures of spoken dialogue systems such as Siri or Cortana. It discusses how humans and computers interact. In particular, it focuses on the mathematics behind how computers decide how to respond to an input which may or may not be human. The dissertation then proposes a method to improve on the limits of the current mathematical model. This method can be used with multiple input modules, such as microphones, webcams or laser sensors. This method is evaluated with a simulation of three input modules. In the evaluation, the time taken to make the decision via this method is one-third of the time needed via current methods.

The following section 1.1 introduces the concept of a dialogue system, and briefly mentions implementations of such systems.

### 1.1 Spoken Dialogue Systems

A spoken dialogue system is a program which communicates with a human through speech, and is also known as a Speech Dialogue System (SDS). Spoken dialogue systems are expanding in use, with common deployments including call centres and voice interfaces such as Apple's Siri. A basic spoken dialogue system is made up of six main components (Jurafsky et al., 2000, p. 857), as seen in Figure 1.1. These are as follows:

- An Automatic Speech Recogniser (ASR), which takes input from the microphone and transcribes it;
- A Natural Language Understanding component (NLU);
- A Dialogue Manager;
- A Task Manager;
- A Natural Language Generator (NLG); and



Figure 1.1: Diagram of a SDS

- A Text-to-Speech synthesiser, which realises the text output from the NLG as sound and plays it through the speakers.

A common deployment of a SDS is in a call centre, to handle basic tasks, such as fielding the call to the correct department. This deployment can make use of a task-based approach that is suitable for rule-based or frame-based systems. These are often based on a single sensor: the verbal phone input. A key problem is the noise that the ASR must deal with, and the resulting uncertainty that can cause difficulties for the dialogue manager.

More recently, spoken dialogue systems have provided an interface with mobile devices, with Apple's Siri being the most recognisable example. Unlike a call centre, a smart device may have multiple types of sensory information available to it, such as a camera; physical orientation; heart rate sensor; etc. Previous multimodal systems, such as the SEMAINE toolkit (Schröder, 2010) force the sensory input to be combined before the information is passed to the dialogue manager. Similarly, dialogue managers to date have required the information to be combined. Young (2010) on Cognitive User Interfaces shows that a Partially Observable Markov Decision Process (POMDP) may be applied to sensors other than ASR, but this requires an individual POMDP for each sensor.

## 1.2 Aims

While spoken dialogue systems are widely implemented, at present they are restricted to a singular input type. Typically, this is automatic speech recognition. The aim of this thesis is to examine how current models may be expanded to handle multiple inputs in modular form. This is done by evaluating a mathematical model called Partially Observable Markov Decision Processes (POMDPs) for tractability, and partitioning it in order to handle modular inputs.

## 1.3 Significance of Study

While the model described in section 3.3 is conceived in the context of spoken dialogue systems, it is not restricted to this field. The POMDP model has been used in other fields for decision making, including for collision detection, healthcare, and conserving endangered species (Temizer et al. (2010), Hauskrecht & Fraser (2000) , Chadès et al. (2008)). Similar problems of tractability occur in other implementations of POMDPs, due to limitations of the model. As such, any improvements in the model itself can be applied to other fields where the model is implemented. In some of these fields,



such as collision detection which requires three-dimensional motion, being able to handle multiple modular inputs is of significant value, as it allows the system to expand beyond current POMDP solver software.

## 1.4 Outline of Thesis

The remainder of this thesis has the following structure. In chapter 2, the recent literature is discussed. This includes literature on human-computer interaction and Partially Observable Markov Decision Processes. Notable problems with the current state of the art are discussed in this chapter. The bulk of chapter 3 focuses on partitioning the POMDP model: why it should be done; and methods for implementing this. Theoretical results of the partitioned model are included here. The chapter also reviews how a POMDP model may be learned from a corpus. The next chapter, chapter 4, describes how the partitioned model was implemented in a Car Advisory System. This occurred due to the lack of a suitable corpus of dialogues. The chapter evaluates the partitioned model against an unpartitioned version, which is implemented using the APPL software (Kurniawati et al., 2008). The final chapter is chapter 5, which reviews the thesis. Any failings in the dissertation are considered, and methods to further evaluate the model are discussed in section 5.1. The appendix contains documentation for the implementation of the partitioned POMDP model as a car advisory system.

## Chapter 2

# Current Literature

This chapter discusses human-computer interaction, and introduces a mathematical model type called Partially Observable Markov Decision Processes (POMDPs). The chapter starts by discussing human-computer interaction, and the motivations for using a dialogue interface. In subsection 2.1.1, the issues that dialogue managers face are discussed, notably ambiguity. The many sources of ambiguity are considered in relation to dialogue managers, and how they handle and resolve uncertainty. Then, section 2.2 formally defines the POMDP model, and is extended by subsection 2.2.1, which discusses the concept of a policy. A policy is determined for a specific POMDP, and directly affects the outcome of the POMDP.

The chapter then summarises the history and motivation of POMDPs as dialogue managers. It looks at how POMDPs have been trained from data in order to produce a dialogue manager. It then looks at the computational complexity of calculating an optimal policy for a given POMDP, and discusses approximation techniques for calculating an optimal policy. Other examples of the POMDP structure used for dialogue management are examined, particularly variations of the POMDP structure, or where it is combined with other techniques.

The chapter then moves on to consider current problems with the state-of-the-art approach. In particular, it considers the issues of when a system has multiple modules of input. The key problem raised here is that of scaling - that each possible combination of states from each of the inputs must have its own state under the standard POMDP model. This results in an extremely large state-space. The final section introduces concepts needed to understand the model which is proposed in the next chapter, in order to address these problems.

### 2.1 Human-Computer Interactions

Human Computer Interaction is the field which looks at how we as humans interact with computers. It studies both the interfaces used, such as mouse, keyboards, screens, etc; but also includes the social and psychological aspects of how a human reacts to computers. Spoken dialogue systems have

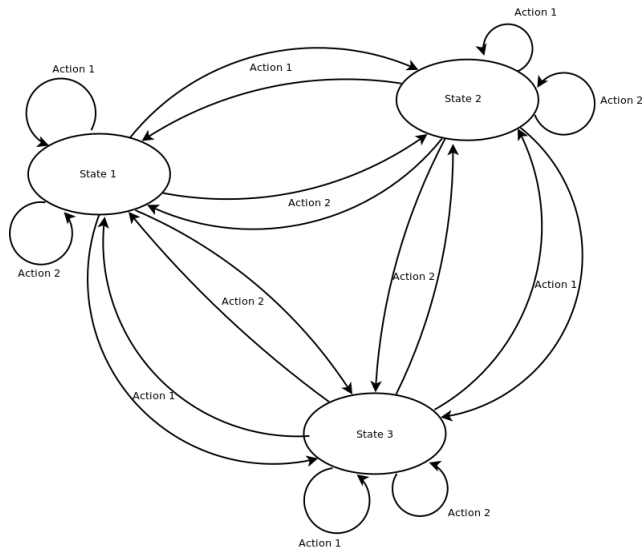


Figure 2.1: Diagram of a Markov Decision Process

been deployed within smart mobile devices, but there is a great deal more to be done within the field of Human-Computer Interaction. While graphical user interfaces (GUIs) have been the base of most human-computer interaction for the past twenty years, they require the visual focus of the user. Moving towards a dialogue interface reduces this dependency on the visual focus, allowing it to be used in more scenarios, such as situations that must be hands-free. The dialogue manager is an important aspect of a SDS. Its role is to decide, given the input from the NLU component, what the output of the NLG should be. Initially, dialogue managers were rule-based systems of if-then statements. Each possible scenario that the system might encounter needed to be coded individually. The simplest type of dialogue manager uses a finite-state machine (sometimes called a finite state automaton, or FSA) as the underlying architecture. In these, there is a flow-chart style diagram which the dialogue runs through, with no variation beyond the paths allowed by the FSA, and no flexibility. In effect, the system controls the conversation. My work makes use of a statistical model, a Partially Observable Markov Decision Process (POMDP), which takes into account input in addition to the ASR. A POMDP is more flexible than a FSA, since it takes into account the possibility that the observed input was flawed. This is a result of the observation function (section 2.2). My model further allows for the input from multiple independent streams, in order to reduce the noise of the observed inputs.

A dialogue manager can also be frame-based. This is still a rule-based system, but has a frame of information that it requires from the user before it can complete the task it is designed for. This makes it more flexible than a finite-state architecture. The system asks questions of the user in order to fill in slots of the frame. With this, the system has some flexibility for the user to direct the conversation, skipping questions as needed. In the early 2000s, there was a move towards statistical dialogue managers. These dialogue managers can have a more dynamic strategy to maximize dialogue success. Early attempts used Markov Decision Processes (MDPs). A MDP is a stochastic process

with a set of states  $S$  that the system may be in; a set of actions  $A$  that the system may take; and a reward function  $r : A \times S \rightarrow \mathbb{R}$ , denoted as  $r(a, s)$ , that rewards the system for taking an action  $a$  from state  $s$ . A state  $i$  is called an absorbing state if, once the system is in state  $i$  it will remain in  $i$ . That is, if  $\mathbb{P}(X_{t+1} = i | X_t = i) = 1$ , then  $i$  is an absorbing state. This means that the system, once in this state, will never leave, as  $\mathbb{P}(X_{t+2} = i | X_{t+1} = i) = \mathbb{P}(X_{t+1} = i | X_t = i) = 1$ , and similarly for all future states. If  $\mathbb{P}(X_{t+1} = i | X_t = i) < 1$ , then the state is not an absorbing state, by definition. Typically, one assumes that the system terminates on absorption. Initialising the MDP will designate one of the states as the start state. The reward is accumulated by the system over each time step, and the final cumulated reward may be viewed as how well the system performed. In order to maximise the total reward accumulated, a policy is used (subsection 2.2.1). A decision rule dictates to the system which action  $a$  to take when it is in a given state  $s$ . A policy is a set of decision rules. Figure 2.1 shows a basic MDP without a reward function, with 3 states and 2 actions available to the system. An example reward function  $R : S \times A \rightarrow \mathbb{R}$  might define the following:  $R(s_1, a_1) = 10$  and  $R(s_1, a_2) = 1$ . This would bias the system towards taking action  $a_1$  from state  $s_1$  over taking action  $a_2$  if large positive values are preferred.  $R$  would be defined for all  $s \in S$  and  $a \in A$ . However, there is still a possibility that the system will take some other action, (determined by the policy and whether short-term or long-term gains are preferred.) What action the system takes is determined by the policy, which can be adjusted to favour short-term or long-term gains. Note that the reward is dependent on the state that the system is in when it takes the action, but independent of the state that the system moves to.

Like the rule-based systems, MDPs rely on the input being dependable, despite the many potential sources of error. The MDP system believes that at each time point, it is in a single specific state, with no possibility of being in a different one. However, this contradicts a key feature of natural human-human dialogue: dialogue is uncertain.

### 2.1.1 Mutual Understanding

A key issue that a dialogue manager needs to handle is the fact that the received input may be flawed. This can cause the dialogue manager to query the entire conversation, rather than identify a single part of an utterance that is causing problems. This uncertainty in the input means that many dialogue managers struggle when an unforeseen situation occurs. This is not a fault due solely to the technology: dialogue between humans is inherently uncertain. This comes partly from the ambiguity inherent in natural language, but also how language is used in conversation. Within dialogue, the meaning of a particular word must be inferred from its context. It must also be recalled later in the conversation, without the use of further disambiguation techniques. A dialogue manager must infer from the observed natural language input what a user intends. This can be easily tripped up when the user intent and the user words are mismatched, such as with sarcasm. The realised words of the user are treated by the dialogue manager as observations, a means to infer information about the state of

mind of the user. The intentions and goals of the user, inferred via the realised words, are treated by the dialogue manager as states. In this way, the dialogue manager attempts to model the goals of the user from the observed input received.

However, ambiguity of language is not the only issue that human-human conversation must handle. In human-human dialogue, a technique called common ground, discussed by Clark (1996), can be used to maintain disambiguation of language, but also handle established conventional meanings within communities of practice. Thus, utterances that are particular to a community or dialect are understood by those belonging to that community, but also serve as markers of identity as members of that community. As such, the knowledge of the community may impact the understanding of the utterance. Analogously for the dialogue manager, this means that the user model that provides input into a dialogue system could affect how an utterance is understood. Hence, the inaccuracy of the user model could introduce uncertainty into the system.

Another source of uncertainty within human-human dialogue is that of the utterer's meaning. Grice (1989) has a theory of utterer's meaning which distinguishes between what is said; what is implicated conventionally; and what is implicated for conversation. The interaction of these types of meaning can lead to uncertainty as to which meaning is actually intended. For a dialogue system, this means that the output of the Natural Language Understanding component may contain a meaning other than what was intended by the utterer. This in turn compounds the uncertainty that the dialogue manager must deal with.

The uncertainty discussed above is inherent to human-human conversation. When a computer is introduced to play the role of one of the participants, further uncertainty is introduced. As mentioned, which meaning is inferred by the Natural Language Understanding component can introduce uncertainty. This uncertainty may be greater than when humans converse, since humans are better at identifying which meaning is implied by the utterer. Other technical sources of uncertainty includes how the input speech signal is processed, which can affect the performance of the ASR. This may also be affected by the accent of the utterer and the amount of background noise.

A key aspect of the POMDP model for dialogue management is how it handles uncertainty. The model expects the inputs (observations) to be uncertain, and the observation function describes how this uncertainty affects the state of the dialogue. Similarly, the model does not assume at any point that it is in any particular state. Instead, it maintains a probability distribution over all possible states, and updates this distribution as new observations are made. This probability distribution over states is known as the belief state. This means that it can jump out of a belief when an observation which contradicts its current belief is received. Within a dialogue manager, the states represent the user intent. This corresponds to what the dialogue manager believes is the state of the dialogue. At each turn, it receives an observation  $o$  which it uses to update the belief state - the distribution over all possible states. The system knows the probability of receiving any particular observation, given the previous state and action taken. This means that the uncertainty of the input is dealt with relative

to what is already known by the system. The system makes its decisions dependent on the belief state, which can be considered as the distribution over user possible user intents, with the highest probability corresponding to the most likely user intent. This handles the uncertainty better, since every input is considered relative to what was previously uttered by the user and by the system, so increasing the robustness of the system. The mathematics of POMDPs are further described below in section 2.2.

## 2.2 Mathematics of POMDPs

A partially observable Markov Decision process (POMDP) is a stochastic process - a mathematical system which varies with respect to time  $t \in \mathbb{N}$ .

For any set  $\Delta \subseteq \mathbb{R}$ , a collection  $\{X_t : t \in \Delta\}$  of random variables is called a stochastic process. Note that a POMDP works in discrete time units, so here one may consider  $\Delta = \mathbb{N}$ . A POMDP is a Markov process, which means that the Markov property holds. This property states that, conditional on the present, the future is independent of the past. Denote the probability of event  $A$  occurring as  $\mathbb{P}(A)$ .

A stochastic process  $\{X_t, t = 0, 1, 2, \dots\}$  is called a Markov process if for non-negative integers  $s, t$  and any collection of states  $(x_0, x_1, x_2, \dots)$  the following probability holds:

$\mathbb{P}(X_{t+s} = x_{t+s} | X_t = x_t, X_{t-1} = x_{t-1}, \dots, X_0 = x_0) = \mathbb{P}(X_{t+s} = x_{t+s} | X_t = x_t)$ . This is known as the Markov Property. Note that the  $X_{t-1} = x_{t-1}, \dots, X_0 = x_0$  term dependencies on the left-hand side are discarded in the right-hand side. This is equivalent to saying that, conditional on the present, the future is independent of the past. Equivalently, what happens tomorrow depends on what happens today. What happens today depends on what happened yesterday, but what happens tomorrow does not depend on what happened yesterday.

A basic diagram of a POMDP can be seen in Figure 2.2. Here, the POMDP has 3 states and 2 actions like the earlier MDP (Figure 2.1), but is faded out because we do not know exactly which state the system is in. The observations are used to form a statistical distribution, represented in the image by the bar chart, which denotes the belief state. The policy is then consulted, and this determines which action should be taken. The action then feeds back into the POMDP for the next time state.

Formally, a POMDP is a tuple,  $\{S, A, T, R, O, Z, \lambda, b_0\}$ , where:

1.  $S$  is a set of states;
2.  $A$  is a set of actions;
3.  $T$  is a transition function,  $T : S \times A \rightarrow S$ ;
4.  $R$  is a reward function,  $R : S \times A \rightarrow \mathbb{R}$ ;
5.  $O$  is a set of observations;

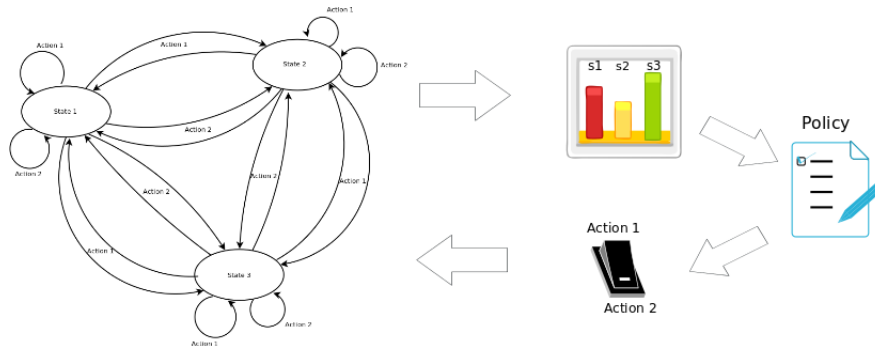


Figure 2.2: Diagram of a POMDP

6.  $Z$  is a observation function;
7.  $\lambda$  is a discount factor; and
8.  $b_0$  is the initial belief state

A POMDP has a set of states  $S$  (item 1) which describe the world in which it operates. However, at any given point in time after initiation of the system ( $t > 0$ ), the state which the system is in may not be known exactly. Instead, the state is inferred from the inputs received about its external world. These inputs are called ‘observations’  $O$  (item 5) and the probability of receiving a certain input  $o$  is considered with respect to the state  $s$ , so allowing the system infer from the observation which state the system is most likely to be in.

However, since this is a probability, the system maintains a distribution over the states. This distribution is referred to as the ‘belief state’ for each time unit.

More formally, the belief state at time  $t$ ,  $b_t$  is a vector of  $m$  dimensions, where  $m$  is the number of states in the system. That is to say,  $b_t = (b_1^t, b_2^t, \dots, b_m^t)$ . Note that for given time  $t$ ,  $\forall i \in S$ ,  $b_i^t \in [0, 1]$ , and  $b_i^t$  is the probability of the system being in state  $i$  at time  $t$ . Thus,  $\sum_i b_i^t = 1$  at all time units  $t$ .

The initial belief state,  $b_0$  (item 8) is predetermined and part of the initialisation process. Thereafter, for time  $t > 0$ , the belief state depends on the initial belief state, the actions taken, and the observations received.

The parameters of the distribution can change with time. For example,  $b_0$  could be the uniform distribution, but at some later time the distribution  $b_t$  might morph to show peaks for particular states.

The POMDP accepts inputs from its external world, which are referred to as observations. In the context of a dialogue manager, these observations may come from ASR, facial recognition, or any sensory input where the signal has been processed. The system can perform actions from the set  $A$  (item 2) which may affect its world. These may include applying a parser, waiting a period of time, or outputting a message to a synthesiser.

At each time unit  $t$ , for the current belief state  $b_t$ , an action  $a$  is taken. In order to determine which action should occur, a decision rule is used. A decision rule dictates what action to take when

in a given belief state. A policy is a tree of decision rules, dictating what the system should do for each belief state (subsection 2.2.1). This tree allows for the different possible observations that might be received by the system, but for a realised sequence of belief states and observations, one sequence of the branching process is actually followed. Typically, the policy is calculated offline ahead of implementing the POMDP, and aims to maximise the total expected reward, as dictated by the reward function (item 4).

It is possible to bias a system towards certain actions by specifying the reward function  $R$  (item 4) to do so. This can be considered as giving the system “points” for taking a certain action from a certain belief state.

For each action, from each state, there is a probability to go to each other state. These probabilities are determined by the transition matrix  $T$  (item 3). Similarly, for each observation and action-state pair, there is a probability of what the next state is. These probabilities form the observation matrix  $Z$  (item 6). The variable  $\lambda$  (item 7) is the discount factor, which affects the reward function. It helps prevent the system from remaining in one (undesired) state or in a loop, if the states involved are not termination or absorbing states. It works by reducing a proportion of the reward given the system at each time step. This works to balance whether the system will favour immediate rewards or long term rewards.

### 2.2.1 Policies

White (1993, p. 26) writes that for Markov Decision Processes “a decision rule  $\delta_t$  for time unit  $t$  determines a probability distribution of actions  $a \in A$  when the history is known ... a policy is a sequence of decision rules. A policy is denoted by  $\pi$  and takes the form  $\pi = (\delta_1, \delta_2, \dots, \delta_t, \dots)$ ” Note that each policy  $\pi$  maps from a belief state to the set of actions  $A$ , i.e.  $\pi : b_t \rightarrow A$ .

Crucially, a policy informs the system what action to take at any time unit of the decision process. In section 2.2, the reward function,  $R : S \times A \rightarrow \mathbb{R}$  was introduced. As stated there, the reward function can be used to bias the system towards certain actions. This is done by calculating an optimal policy, which is one which maximises the expected total reward.

For the first  $n$  time units, let  $\Theta$  denote the cumulative discounted reward, where  $r_t$  is the realised reward,  $\Theta = \sum_{t=1}^n \lambda^t r_t$ . The expected return,  $\mathbb{E}(\Theta)$  is dependent on the realised reward,  $r_t$ , which in turn depends on the belief state and on the policy  $\pi$  which dictates which belief states may realistically be achieved. Note that not all belief states may be achieved from the initial belief state,  $b_0$ . Only those belief states which may, through a finite sequence of actions be realised, are achievable. Further, as some actions and observations are more probable than others, some belief states are significantly unlikely to occur.

For a given policy, the infinite horizon expected reward as a function of the belief state is called the value function. This is calculated as:



$$V^\pi(b) = \sum_{t=0}^{\infty} \lambda^t r(b_t, a_t) = \sum_{t=0}^{\infty} \sum_{s_i \in S} b_i^t r(s_i, a) = \sum_{t=0}^{\infty} \sum_{s_i \in S} b_i^t r(s_i, \pi(b_t)),$$

where  $b_t$  is the distribution over all states at time  $t$ , and  $b_i^t$  is the probability of the system being in state  $s_i$  at time  $t$ .

The performance of the system is dependent on the policy, as the value function depends on the policy. An optimal policy  $\pi^*(b)$ , where  $\pi^*(b) \in A$ , is one that maximises the value function. Within a spoken dialogue system, a POMDP dialogue manager will perform best when its policy is optimal. Thus, the computational tractability of calculating the optimal policy is relevant. Papadimitriou & Tsitsiklis (1987) demonstrated that calculating the optimal policy exactly is PSPACE-hard, dependent on the state-space  $|S|$ . This places the problem in the same complexity class as several other problems, listed by Garey & Johnson (1979), such as the equivalence problem for non-deterministic finite state automata (determining whether two automata  $A_1, A_2$  accepting the same input alphabet  $\Sigma$ , where  $\Sigma \geq 2$  recognise different languages). More importantly, it is in the NP complexity class. Algorithms exist to calculate the optimal policy exactly, such as Kaelbling et al. (1998), but these swiftly become impractical when  $|S| + |O| \geq 23$ , which took 2 hours to calculate on machines in the 1990s (Littman et al., 1995). This is because the belief space grows exponentially. However, the lack of tractable exact algorithms does not prevent POMDPs being used as dialogue managers. A number of algorithms exist to calculate an approximation to the optimal policy, beginning with the Point Based Value Iteration algorithm introduced by Pineau et al. (2003), which was applied to a POMDP with 870 states, 5 actions and 30 observations. Approximation algorithms restrict the belief space to reachable belief states – those beliefs that are possible from the current belief state. More recent algorithms include the SARSOP algorithm, introduced by Kurniawati et al. (2008). Work in reinforcement learning has also looked at ways to improve the way the reward function is defined. More details can be found in section 2.3. Whilst the policies learnt from approximation algorithms are not the exact optimal policy, they are usable within dialogue managers. Thomson et al. (2007) adapted a technique for MDPs, Q-learning, to approximate a policy for a tourist information domain dialogue manager. Within this domain, the learnt policy successfully completed 90.6% of the 160 dialogues. The main drawback of the Q-learning technique is the several thousand dialogues needed to train the policy initially. Typically, this learning needs to be done in advance, but may be combined with reinforcement learning for further improvement.

Whilst I will not be calculating an optimal policy, I will use the SARSOP algorithm in order to approximate a policy for my POMDP model.

## 2.3 Literature Review

The history of POMDPs is a long one. They originated out of operations research, in particular the design of control systems (Åström, 1965), as an expansion of Markov Decision Processes. The motivation then was to handle situations when the state variables could not be measured, and were not completely observable. This applies to a number of situations, dialogue as one of them. Outside the field of human-computer interaction, POMDPs have been successfully implemented in a variety of fields. These include collision detection in unmanned aerial vehicles, (Temizer et al., 2010); in treatment of heart disease, (Hauskrecht & Fraser, 2000); and conserving endangered species, (Chadès et al., 2008). Having mentioned these other areas which make use of POMDPs, the rest of this section will focus on the history of POMDPs within dialogue systems.

The use of Partially Observable Markov Decision Processes (POMDPs) as dialogue managers began with the paper by Williams & Young (2007). This paper gave examples of a POMDP being used as a statistical model base for a dialogue manager and marked a shift towards statistical dialogue managers with POMDPs providing the core model. The motivation for this was to handle the input from automatic speech recognisers which were, and remain, unreliable. The main advantage of using a POMDP is that the model takes into account the notion that the observations received may be uncertain, as it considers the probability of receiving that observation. A POMDP gives more flexibility over the more tractable Markov Decision Process (MDP) which assumes that the exact state is known exactly at all times. The motivation was built upon further by Young et al. (2010), which described the mathematics of POMDPs. Like Williams & Young (2007), Young et al. (2010) model the user intent as the states of the POMDP, an approach which was followed by later work, including Chinaei & Chaib-draa (2011) and Yoshino & Kawahara (2014).

The literature that builds on Williams & Young (2007) has been in three broad areas: the training of states, actions, observations and transition function from a corpus; the problem of approximating policies; and implementing POMDP models as dialogue managers.

The work of Chinaei et al. (2009) and Chinaei & Chaib-draa (2011) focussed on the training of a POMDP from dialogue data. Chinaei et al. (2009) looks at learning ‘user intentions’ from data. To do this, the authors apply Hidden Topic Markov Models (HTMM), introduced by Gruber et al. (2007) to dialogues. An HTMM combines Latent Dirichlet Allocation with a hidden Markov model, in order to learn topics from a piece of text. Chinaei et al. (2009) apply this to dialogues, with the attitude that these topics learnt from the HTMM correlate to user intent. Latent Dirichlet Allocation is a probabilistic model to generate collections from discrete data, where each collection is viewed as a mixture over a finite set of topics (Blei et al., 2003). Chinaei & Chaib-draa (2011) use the topics learnt from the HTMM as the states of a POMDP. This is coherent with the approach of Williams & Young (2007), where the states of a POMDP represent user goals. The approach of Chinaei & Chaib-draa (2011) is that the topics represent the user goals, which in turn represent the states of the

POMDP.

Papadimitriou & Tsitsiklis (1987) demonstrated that calculating the optimal policy with respect to the reward function is intractable; more specifically, it is *PSPACE*-hard. However, the Point-Based Value Iteration (PBVI) algorithm, introduced by Pineau et al. (2003) provided the first algorithm that approximated an optimal policy, allowing it to scale to real-world-sized problems. Point Based Value Iteration (PBVI) selects a number of points in the belief space and calculates the value function of these points. It maintains a backup operator in order to calculate the value function at time  $t$ . This backup operator is then modified to retain a single  $\alpha$ -vector for each belief point. Later algorithms include the SARSOP algorithm described by Kurniawati et al. (2008) which adapted PBVI to optimally reachable belief spaces; and adaptations of Q-learning for MDPs Thomson et al. (2007) were evaluated against PBVI. (Kurniawati et al., 2008, p. 2) describe PBVI as “the first point-based algorithm that demonstrated good performance on a large POMDP called Tag, which has 870 states.” This particular POMDP was based in the robotic domain and was inspired by the game of lasertag. Kurniawati et al. (2008) also mention two other point-based algorithms, HSVI2 and FSVI, on which their SARSOP algorithm is based. Thomson et al. (2007, p.11) offer an alternative approach for approximating a policy that differs from the point-based approaches discussed earlier. Instead, the approach is based on Q-learning, which the authors describe as “a technique for online learning traditionally used in an MDP framework” that is in the Monte-Carlo style iterative algorithm.

However, the method of defining the reward function in Williams & Young (2007) was manual. Conservatively, actions that were desired were given a positive reward, and those that were not were discouraged with negative values by the reward function. Frequently, in a real world scenario, the reward function is unknown, but observable (Cao & Guo, 2004). However, whilst reward functions which mimic the observed reward are of interest, since they affect the policy and the behaviour of the POMDP, they are not the focus of discussion here.

One approach to handling the many different concepts that are combined in dialogue managers is in the work of Vargas et al. (2011). In this work, a different POMDP was implemented for each task, each with its own policy. A task-structure controller was used to determine which POMDP was active. This controller was a AND-OR graph with a common root node - effectively a strict hierarchy of multiple POMDPs. This hybrid structure, whilst relatively tractable, requires that the controller has understood the ASR correctly in order to pass control to the correct POMDP. This negates the motivation of using a POMDP: that what is understood about the world is known to be a flawed understanding.

Similarly, Yoshino & Kawahara (2014) use a drop-through style controller to determine the topic that the system should offer, before handing control to the relevant POMDP. This lacks the flexibility to return to earlier topics that are not related to the topic in focus. Here, tractability is gained through a divide-and-conquer approach of multiple small POMDPs; but this is at the expense of the flexibility that the POMDP offers.

Others have looked at partitioning a POMDP, notably Hsiao et al. (2007). However, their work focussed on moving a robot around a two dimensional space, not a speech dialogue system. Also, while they use the term ‘partition’, this is in reference to regions of a single state space where the probability of being in that region is high. This affects the belief state, which may have a high certainty of being in a particular region. However, in this scenario, there is only one input type, which is the contact status of the robot sensor. As such, whilst dividing the belief state into such regions has computational advantages, the concepts in this paper are different to those presented below for maintaining a partition for each feature set.

## 2.4 Current problems

As section 2.3 explains, whilst POMDPs have been used as dialogue managers since 2007, there remains a problem of tractability when the model is scaled to a large state space. Attempts to handle this have formed hybrid systems, but these fail to take full advantage of the flexibility and uncertainty handling that the POMDP provides. Hybrid systems require that the correct POMDP is chosen by a rule-based controller, which struggle with faulty ASR.

One example of a hybrid system is Wu et al. (2015), which used a POMDP dialogue manager as a means of knowledge acquisition via slot-filling. To better handle the uncertainty introduced by the ASR, they used an N-best list as a distribution of the observations. In their work, they used humans to teach the system properties of various objects. They divided the 128 user goals into four sets, and noted that  $|S_u|$  is the size of the Cartesian product of the four sets (six sets of two elements each, and a further two sets of one element each.) They consider a total of 3841 states,  $s = (s_u, a_u, s_d)$  plus an absorbing state, where  $s_d$  is the dialogue history, and  $a_u$  is the user action. The POMDP agent has 134 actions available to it.

In the research on dialogue managers to date, there have been two main approaches: hand-crafted finite-state automata (FSA), and the statistical approach of MDPs and POMDPs (Williams & Young (2007), Young et al. (2013), Thomson et al. (2007)). It certainly has advantages over previous rule-based models such as FSAs or slot-filling models due to its flexibility, and over the more tractable Markov Decision Process model due to its handling of uncertain input.

More recently, there has been interest in combining the advantages of both hand-crafted FSAs and statistical approaches. An example of this is Lison (2015), who considers probabilistic rules, which inserts expert knowledge into the POMDP system by placing if-then conditionals on the transition models. This states that when a predefined condition holds, then the distribution over possible effects is defined in a particular manner. Formally, a probability rule is an ordered list of branches,  $[br_1, br_2, \dots, br_n]$ , where each branch  $br_i$  is a couple of a condition  $c_i$  and a probability distribution over possible effects.  $\mathbb{P}(E_i)$ . This means that the conditions can be hand-crafted, but the probability distributions remain, to improve how the model handles uncertainty. Lison (2015) evaluated this model

against a hand-crafted FSA and against a POMDP trained on a Wizard-of-Oz system. The task was for the human user to instruct a Nao robot to traverse a maze with imaginary rules and to manipulate a graspable object. The dialogues evoked by the three systems were evaluated on 15 metrics, of which 9 were objective and 6 subjective. The objective evaluations included average number of repetitions, average number of turns, and overall dialogue duration. The subjective measures were the multiple answer questions from the user survey.

However, in expanding dialogue managers to handle multiple tasks, the POMDP does not make best use of all the information that is available to it. Currently, dialogue managers have any multi-modal features combined into a vector before the POMDP model receives the state. That is, for each feature  $f \in F$  and  $g \in G$ , the multimodal feature is treated as the vector  $(f, g)$ . This necessitates a large state space, because every possible combination of features requires its own state. In turn, this makes updating the belief function less tractable, since to update the belief function, one must calculate for each state the probability of being in that state. Thus for  $F_i, i \in I$  where  $I$  is an index set (a set used to label the collection of feature sets) and  $F_i$  is a set of features, the state space needed by the POMDP is  $\prod_{i \in I} |F_i|$ . This means that scaling remains an issue. Due to the large state space required, sensitivity to a given feature within a state is reduced, because it competes with other states which have the same feature in alternative combinations.

In the model I propose below, the state space remains partitioned by the input features. The overall state space can be considered as a function of these partitions. This means that the number of states over which the system must maintain a belief distribution is reduced, since each input feature set is considered with its own distribution. Thus, the uncertainty of each input type is independent from the uncertainty from other input types. Wherever there are actions which do not affect at least one input type, there is a reduction in the number of calculations needed to update the belief state. This increases overall tractability.

By retaining the partitions of the state space, the sensitivity to different features increases, since each distribution is over a smaller number, so any deviation from the expected distribution is more noticeable. Also with these partitions of the state space, the weightings on the feature sets can be dynamic, adjusted depending on what information comes from each input feature. As a result of the increased sensitivity, the actions could be adjusted to target a single feature set, and the POMDP policy can be adjusted to reflect this. This model is a version of a POMDP, which means that unlike the hybrid systems described earlier, it retains the flexibility to move between states in the partition. Also, it handles the uncertainty of the inputs because it does not require a rule-based controller to determine whether a partition should be in focus.

The intuition behind this is that knowing the component pieces gives information about the combination of them. For example, consider a course that is assessed via continuous assessment (component  $C$ ) and via exam (component  $E$ ). The function to combine them into a final score here is a weighting function,  $f(c, e) = \lambda c + (1 - \lambda)e$ , where  $c \in C, e \in E$  and  $\lambda \in [0, 1]$ . Thus, to know each of  $c, e$  implies

knowing  $f(c, e)$ . The reverse may not be. That is to say, knowing  $f(c, e)$  does not give information about  $c$  or  $e$ . Also, if a distribution over  $C$  is available, then identifying deviation of  $c$  from the distribution is possible. Likewise with  $E$ . Further, it is possible to identify deviation between the distributions over  $C$  and  $E$ .

Typically with dialogue managers, the combination function is as a vector of the multimodal inputs,  $f(x, y) = \langle x, y \rangle$ , for some inputs  $x, y$  from input types  $X, Y$ . This means that the order of the range of the function  $f$  is  $|X||Y|$ , since  $f : X \times Y \rightarrow X \times Y$ , and  $|X \times Y| = |X||Y|$ . If instead, we consider each input types  $X, Y$  separately, with a distribution over each, then there is a distribution over  $|X|$  states and a distribution over  $|Y|$  states, which together is two distributions over  $|X| + |Y|$  states. This is smaller than  $|X||Y|$ . If the number of states over which the system needs to calculate a distribution in order to update the belief state, then the system should be more tractable.

## Chapter 3

# Partitioning the POMDP

This chapter briefly looks at how a POMDP model can be trained from data, based on the literature and my own work to apply the method to a different corpus. The motivation for training a POMDP from data is that it makes the states of the POMDP directly correlate to the corpus it is trained on, rather than being formed by handcrafting. The chapter then looks at motivation for partitioning a POMDP for modular input. The main motivation is to reduce computational costs. Partitioning the POMDP allows the total number of states to be reduced. This reduces the costs associated with simulating a POMDP. A secondary motivation is to reduce the noise in the system. Partitioning maintains a separation of the input methods. This means that the system can use each input type independently. The system can also use the alternative input types to resolve uncertainty. For example, the separation allows the system to detect when there are discrepancies between the input types. These discrepancies could hold further information, such as when ASR and prosodic information are in conflict may indicate sarcasm.

### 3.1 Learning a POMDP

Hidden Topic Markov Models (HTMMs) were introduced by Gruber et al. (2007) for modelling the words in a text corpus as topics, by combining Latent Dirichlet Allocation with Hidden Markov Models. Chinaei et al. (2009) adapted this method to dialogue corpora, with the motivation that the topics which are learnt by the HTMM are correlated to the intention encoded within user utterances. Chinaei & Chaib-draa (2011) apply this method to the SACTI corpus of wizard-user dialogues. Here, learning the topics via an HTMM to represent the user intention, the topics are then used as the states of a POMDP, in accordance with Williams & Young (2007), where the states represent the user ‘goal’ or intent.

I applied the OpenHTMM toolkit to the Edinburgh Maptask corpus, aiming to identify five topics. The toolkit was initially run to identify ten topics, but this clustering, when examined, seemed incoherent. With five topics, one cluster was not clearly coherent as the other four, but was more

successful than with four topics. A naive Bayes classifier was used to assign topics to utterances. The dialogue moves that were included in the annotated maptask were treated as the actions and the observations when performed by the follower and the guide respectively. These were aligned with the utterances in order to calculate the transition matrix and the observation matrix. The entries in the matrix was calculated as follows:  $T(s_1, a_1, s_2) = \frac{\text{count}(s_1, a_1, s_2) + 1}{\text{count}(s_1, a_1) + K}$  where  $K = |S|$ . This is adjusted from Chinaei et al. (2009) in order to ensure that each row of the transition matrix sums to 1, which the APPL software by Kurniawati et al. (2008) checks prior to approximating a policy.

## 3.2 Motivation for Partitioning

Having trained a POMDP on dialogue data, one questions whether it is possible to expand this model and training to multiple modular inputs. When humans interact, there is more than what words are said; there is information in how they are said. An effective dialogue system should be able to handle this extra information. Recall that Figure 1.1 represents a SDS, and note that a POMDP dialogue manager is designed to only handle a single input type: from the ASR. As areas such as gesture recognition and facial recognition improve, these provide other input streams which might be used to reduce the noise from the ASR input. These streams will also have inherent uncertainties, but these uncertainties may be different from those of the ASR. The model I propose handles these multiple input streams within the dialogue manager itself, and is designed to accept any number of input types, not just the single ASR input currently accepted by most current dialogue managers. This can be shown in Figure 3.1.

These additional input streams can be used to reduce the noise which the POMDP dialogue manager has to handle. My model maintains these streams as separate input types, independent of the other input streams. This is in contrast to combining the input prior to the dialogue manager, as is customary. By combining the input sets in this way, as a function of the input streams, the number of possible observations is the Cartesian product of all possible inputs. Thus, if there are 20 possible prosodic classes, (set  $|P| = 20$ ), and 15 possible recognised gestures (set  $|G| = 15$ ), then there would be 300 possible input observations over which the POMDP must maintain a distribution (denote  $T = P \times G, |T| = 300$ ). Since solving a POMDP exactly is already intractable, handling additional input streams is computationally difficult, despite the potential for noise reduction. My model maintains the streams as separate, independent input types, and instead calculates a distribution over each input type. Thus, instead of a single distribution over 300 observations, the system maintains two distributions: one over the 20 prosodic classes, and one over the 15 recognisable gestures.

Since  $T$  is a function of  $P$  and  $G$ ,  $\forall t \in T, t = f(p, g)$ , where  $p \in P$  and  $g \in G$ . This means that knowing  $p, g$  implies knowing  $t$ . Given sufficient data, the function  $f$  may be approximated if necessary. This means that considering the sets  $P$  and  $G$  separately will be no worse than considering  $T$  alone. Also, as  $|P|$  is much smaller than  $|T|$ , identifying deviation from the distribution over  $P$



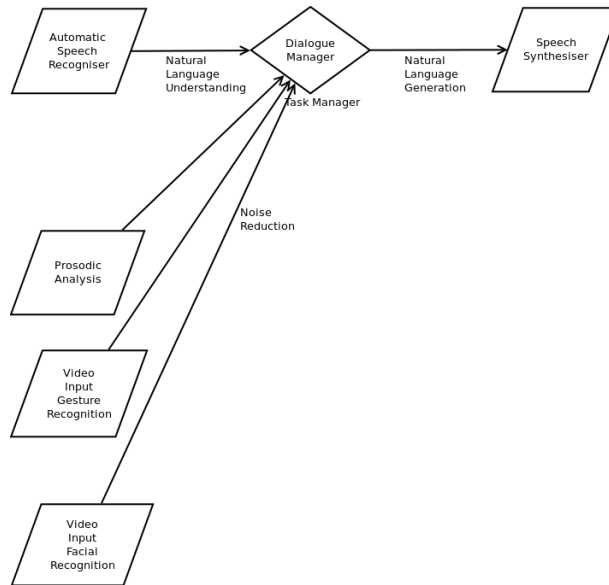


Figure 3.1: Diagram of a SDS

is easier than identifying deviation from the distribution over  $|T|$ . Weightings on  $P$  and  $G$  can be dynamic and adjusted in real time if needed. This means that the system can adapt for any variance in quality from  $P$  and  $G$  in real time. In addition, fewer calculations need to be performed at each stage, since the two distributions over  $P$  and  $G$  have  $|P| + |G| = 35$  probabilities to be calculated at each time step, instead of  $300 = |T| = |P| \times |G|$ . In general, for  $x_i \in \mathbb{N}$ ,  $\sum_{j=1}^n x_i < \prod_{j=1}^n x_i$ . Here, this means that even if the input sets are large, this model will be more tractable than one where the inputs are combined, since the number of calculations that need to be performed at each time step is reduced.

Whilst the model describes feature sets in the abstract sense, these should be defined by the input streams. Thus, if the facial recognition allows for  $f_n$  possible expressions, then that should form one partition, of size  $f_n$ .

For example, within a dialogue system, the feature sets could be facial recognition, prosodic analysis and ASR. When one is lacking meaningful data, the POMDP could adjust its action to focus on the other input type which is offering meaningful data. This could, in turn, affect the user engagement with the dialogue system, because the POMDP is sensitive to more than just the words that are said, but to the other input types as well. For example, discrepancies between prosodic information and ASR could potentially indicate sarcasm. The system could choose actions which focus on one (or more) of the particular input types, and aim to elicit information via a particular input source.

### 3.3 Partitioning the POMDP state space

#### 3.3.1 Defining the model

Let  $S$  represent the set of states of a POMDP, and denote the size of  $S$  as  $n$ , i.e.  $|S| = n$ . Let  $s \in S$  represent a user intention, or goal, that has been identified from the input streams of a spoken dialogue system.

Suppose that  $S$  can be partitioned into “domains”, where  $S$  is a function of those domains. Suitable domains would be separate input streams, such as ASR and facial recognition. Denote these domains as  $\tilde{S}_i$  where  $i \in I$  is a finite indexing set, with  $|I| = k$ . Then  $S = f(\tilde{S}_1, \tilde{S}_2, \dots, \tilde{S}_k)$ . If  $\forall i \in I$ , the  $\tilde{S}_i$  are independent, then  $f$  may simply be the product of  $\mathbb{P}(\tilde{S}_i)$ . Otherwise,  $f$  may represent a Bayesian network of the  $\tilde{S}_i$ . Further, for any  $s \in S$ , there is a unique domain  $\tilde{S}_i$  to which each state  $s$  belongs. This is equivalent to stating that for  $i, j \in I$  such that  $i \neq j$  if  $s \in \tilde{S}_i$  then  $s \notin \tilde{S}_j$ . This forms a partition of  $S$ , since the  $\tilde{S}_i$  are disjoint and the  $\tilde{S}_i$  cover  $S$ . Where possible, there should be some link between members of a domain, e.g. from the same input stream; within the same semantic category; a domain of topics; a domain of dialogue type.

Note that  $S = \bigcup_{i \in I} \tilde{S}_i$ . For each  $\tilde{S}_i$ , let  $n_i$  denote the size of  $\tilde{S}_i$ . I.e.  $n_i = |\tilde{S}_i|$ . Note that the partition sizes need not be equal. Thus, without loss of generality, one can order the  $\tilde{S}_i$  by size, such that  $n_i \leq n_{i+1}, \forall i \in I$ . Also, note that the following holds, since the sets  $\tilde{S}_i$  are disjoint (each  $s$  belong to a unique  $\tilde{S}_i$ ) and cover the set  $S$ :

$$n = |S| = \left| \bigcup_{i \in I} \tilde{S}_i \right| = \sum_{i \in I} |\tilde{S}_i| = \sum_{i \in I} n_i. \quad (3.1)$$

Let  $\tilde{b}_i(t)$  denote the belief probability distribution at time  $t$  over the class  $\tilde{S}_i$ . This  $\tilde{b}_i(t)$  will have  $n_i$  dimensions, since  $|\tilde{S}_i| = n_i$ . We can consider the vector of the  $\tilde{b}_i(t)$ s as representing a multivariate distribution. Thus, the realised belief at time  $t$ ,  $b(t)$  is a random vector which takes values from the  $\tilde{b}_i(t)$ s. The initial belief at time  $t = 0$  is  $b(0) = \langle \tilde{b}_1(0), \tilde{b}_2(0), \dots, \tilde{b}_k(0) \rangle$ .

Suppose that the observations can be classified in real time into the same “domains” as the states. Then an observation at time  $t$  can be represented as  $o(t) = \langle \tilde{o}_1(t), \tilde{o}_2(t), \dots, \tilde{o}_k(t) \rangle$ , where for some  $i \in I$ ,  $\tilde{o}_i(t)$  is not null. Similarly to the belief function, these observation sets  $\tilde{o}_i(t)$  each form a distribution over the one of the observation domains. Likewise, the realised observation  $o(t)$  can be viewed as a random vector with components from the  $\tilde{o}_i(t)$ . For any  $i \in I$  that  $\tilde{o}_i(t)$  is null, then set  $b_i(t+1) = b_i(t)$ . Otherwise, use an update function as defined below to calculate  $b_i(t+1)$ , where  $b_i(t+1)$  is a function of  $b_i(t), a(t)$  and  $\tilde{o}_i(t)$  for  $a(t)$  the realised action.

For each action  $a \in A$ , let  $T_a$  denote the transition function over the entire state space  $S$ . Let  $P_a$  denote the  $n \times n$  matrix, where the states  $s \in S$  are grouped according to their domain  $\tilde{S}_i$ . Further assume that the  $\tilde{S}_i$  are ordered by size, such that  $|\tilde{S}_i| = n_i \leq n_{i+1} = |\tilde{S}_{i+1}|, \forall i \in I$ .

If action  $a$  has no effect on domain  $\tilde{S}_i$ , then for this time unit  $t$  the domain  $\tilde{S}_i$  should be treated

as a hidden markov model, and updated accordingly when observation  $o_i(t)$  arrives.

If action  $a$  affects only one domain and is closed under that domain,  $\tilde{S}_i$  (that is,  $T_a(\tilde{S}_i) \subseteq \tilde{S}_i$ ), then the function  $T_a$  can be represented as a square  $n_i \times n_i$  matrix existing somewhere along the diagonal of  $P_a$ . This is because the probability of  $T_a(s_j, s_k) = 0$  where  $s_j$  or  $s_k \notin \tilde{S}_i$ . As such, only entries in the matrix  $P_a$  where both the row and the column component represent an  $s_i \in \tilde{S}_i$  will be non-zero. This includes the entries for  $T_a(s_i, s_i)$ , which are denoted along the diagonal of  $P_a$ . Similarly, if  $s_i, s_{i+1} \in \tilde{S}_i$ , then  $T_a(s_i, s_{i+1})$ , denoted just off of the diagonal of  $P_a$ , may be non-zero.

If action  $a$  affects only domains  $\tilde{S}_i$  and  $\tilde{S}_j$  and is closed over those domains, such that for any other domain  $\tilde{S}_k$ ,  $\nexists s_k \in \tilde{S}_k, k \neq i, j$  such that  $T_a(s_i) = s_k$  nor  $T_a(s_j) = s_k$ , then for each element  $s_i \in \tilde{S}_i$  and  $s_j \in \tilde{S}_j$ ,  $T_a$  denotes the function  $T : A \times \tilde{S}_i \times \tilde{S}_j \rightarrow \tilde{S}_i \times \tilde{S}_j$  for  $a \in A$ . Without loss of generality, assume  $n_i \leq n_j$ , since the  $\tilde{S}_i$ s are ordered by size. As before, we can consider the matrix  $P_a$ . Now, the matrix may only have non-zero entries wherever  $s_i \in \tilde{S}_i \cup \tilde{S}_j$ . If action  $a \in A$  is not closed under one domain, e.g.  $T_a : \tilde{S}_i \rightarrow \tilde{S}_j$ , then it should be treated as though it affects multiple domains,  $T_a : \tilde{S}_i \times \tilde{S}_j \rightarrow \tilde{S}_i \times \tilde{S}_j$ .

Note that the reward function  $R : A \times \bigcup_{i \in I} \tilde{S}_i \rightarrow \mathbb{R}$  need only be defined where  $T_a : A \times \bigcup_{i \in I} \tilde{S}_i \rightarrow \bigcup_{i \in I} \tilde{S}_i$  is defined. This is because where  $T_a : A \times \bigcup_{i \in I} \tilde{S}_i \rightarrow \bigcup_{i \in I} \tilde{S}_i$  is undefined, the probability of taking action  $a$  from  $\bigcup_{i \in I} \tilde{S}_i$  is 0. Thus, defining the reward function where  $T_a : A \times \bigcup_{i \in I} \tilde{S}_i \rightarrow \bigcup_{i \in I} \tilde{S}_i$  is undefined would be to define a function which will never be used.

### 3.3.2 Update Function

The belief state for each  $\tilde{b}_i(t+1)$  over domain  $\tilde{S}_i$  for given action  $a$  should be updated in the following manner:

1. if the realised action  $a$  has no effect on domain  $\tilde{S}_i$  and observation  $o_i(t)$  is null, then set  $\tilde{b}_i(t+1) = \tilde{b}_i(t)$ .
2. if the realised action  $a$  has no effect on domain  $\tilde{S}_i$  and observation  $\tilde{o}_i(t)$  is not null, then update  $\tilde{b}_i(t+1)$  dependent on  $\tilde{b}_i(t)$  and  $\tilde{o}_i(t)$ , like a Hidden Markov Model (HMM)

3. if the realised action  $a$  affects the domain  $\tilde{S}_i$  and observation  $\tilde{o}_i(t)$  is received, then update as a POMDP update function,  $\tilde{b}_i(t+1)$  for each element  $s_i \in \tilde{S}_i$ , with

$$\mathbb{P}(s_i(t+1) | \tilde{o}_i(t), \tilde{b}_i(t), a) = \frac{1}{\mathbb{P}(\tilde{o}_i(t) | a, \tilde{b}_i(t))} \mathbb{P}(\tilde{o}_i(t) | s_i, a) \sum_{s'_i \in \tilde{S}_i} \mathbb{P}(s_i | a, s'_i) b_{s'_i}(t).$$

Here,  $s'_i$  is used to identify the other states in  $\tilde{S}_i$ , and where  $s'_i$  is the state at time  $t+1$ , and  $s$  is the state at time  $t$ .

4. if the realised action  $a$  affects the domain  $\tilde{S}_i$  and no observation is received, then substitute in a null observation and update the belief state according to the distribution of states  $s_{i+1}$  that can occur after action  $a$ .

### 3.3.3 Dependent Partitions

Thus far, the model described has assumed independence of the partitions. However, it is likely within the scenario of a dialogue system that the observed inputs are not independent. Thus the previously mentioned function  $f$ , which links the state space  $S$ , with the partitioned domains  $\tilde{S}_i$ ,  $S = f(\tilde{S}_1, \tilde{S}_2, \dots, \tilde{S}_k)$ , for indexing set  $I, k = |I|$ , and its calculation of the probability,  $\mathbb{P}(S)$  from the probabilities of the domains  $\tilde{S}_i$ , may be a Bayesian network, instead of the product of the probabilities.

Consider also when an element  $x \in \tilde{S}_1 \Rightarrow y \in \tilde{S}_2$ , or when  $x \in \tilde{S}_1 \Leftrightarrow y \in \tilde{S}_2$ :

Let  $x \in \tilde{S}_1$  and  $y \in \tilde{S}_2$ . Suppose  $x \Rightarrow y$ . Then  $\mathbb{P}(y|x) = 1$ .

$x$	$y$	$\Rightarrow$	$\mathbb{P}(y x)$
T	T	T	1
T	F	F	-
F	T	T	$1 - \mathbb{P}(\neg y \neg x)$
F	F	T	$1 - \mathbb{P}(y \neg x)$

This is a dependency of  $\tilde{S}_2$  on  $\tilde{S}_1$ . When  $x \Leftrightarrow y$ ,  $\mathbb{P}(y|x) = 1$  and  $\mathbb{P}(\neg y|\neg x) = 1$ . Also,  $\mathbb{P}(\neg y|x) = \mathbb{P}(y|\neg x) = 0$

Consider the scenario where there are three sensor inputs: one to observe a yes/no answer; one to observe the presence or absence of a smile; and one to measure temperature. Each sensor is assigned to a domain  $\tilde{S}$ . Let  $\tilde{S}_y$  denote the domain of a yes/no answer. Let  $\tilde{S}_s$  denote the domain of presence/absence of a smile. Let  $\tilde{S}_T$  denote the domain of temperature. Then, the update function described in subsection 3.3.2 can be modified for those partitions which are not independent.

Suppose that  $\tilde{S}_s$  is dependent on  $\tilde{S}_y$ , and that  $\tilde{S}_T$  is independent of both  $\tilde{S}_y$  and  $\tilde{S}_s$ .

Then to update  $\tilde{S}_T$  and  $\tilde{S}_y$  is as described above in subsection 3.3.2, but to update  $\tilde{S}_s$  requires the following modifications:

1. if the realised action  $a$  has no effect on domain  $\tilde{S}_s$  and observation  $o_s(t)$  is null, then for each  $s \in \tilde{S}_s$  and  $s' \in \tilde{S}_s \cup \tilde{S}_y$ , set  $\mathbb{P}(s(t+1)|s'(t), a) = \mathbb{P}(s(t)|s'(t-1), a)$
2. if the realised action  $a$  has no effect on domain  $\tilde{S}_s$  and observation  $\tilde{o}_s(t)$  is not null, then update  $\tilde{b}_s(t+1)$  dependent on  $\tilde{b}_s(t) \cup \tilde{b}_y$  and  $\tilde{o}_s(t)$ , like a Hidden Markov Model (HMM)
3. if the realised action  $a$  affects the domain  $\tilde{S}_s$  and observation  $\tilde{o}_s(t)$  is received, then update as a POMDP update function,  $\tilde{b}_s(t+1)$  for each element  $s_i \in \tilde{S}_s$ , with
$$\mathbb{P}(s_i(t+1)|\tilde{o}_s(t), \tilde{b}_s(t), a, \tilde{b}_y(t)) = \frac{1}{\mathbb{P}(\tilde{o}_s(t)|a, \tilde{b}_s(t), \tilde{b}_y(t))} \mathbb{P}(\tilde{o}_s(t)|s_i, a) \sum_{s'_i \in \tilde{S}_s \cup \tilde{S}_y} \mathbb{P}(s_i|a, s'_i) \tilde{b}_{s'_i}(t).$$
4. if the realised action  $a$  affects the domain  $\tilde{S}_i$  and no observation is received, then substitute in a null observation and update the belief state according to the distribution of states  $s_{i+1}$  that can occur after action  $a$ .

### 3.3.4 Theoretical Results

To calculate the updated  $\tilde{b}_i(t+1)$  over  $\tilde{S}_i$ , with  $|\tilde{S}_i| = n_i$  for a given  $a$ , requires calculating  $\tilde{b}_i(t+1)$  for each  $s \in \tilde{S}_i$ . Since there are  $n_i$  elements of  $\tilde{S}_i$ , to update  $\tilde{b}_i(t+1)$  requires  $n_i$  calculations when  $\tilde{S}_i$  is independent. For each  $s \in \tilde{S}_i$  this is, at most,

$$\tilde{b}_{s_i}(t+1) = \left( \frac{1}{\mathbb{P}(\tilde{o}_i(t)|a, \tilde{b}_i(t))} \mathbb{P}(\tilde{o}_i(t)|s_i, a) \sum_{i \in I} \sum_{s_i \in \tilde{S}_i} \mathbb{P}(s'_i|a, s_i) \tilde{b}_{s_i}(t) \right) \quad (3.2)$$

This may be reduced whenever there is a null observation for one of the  $\tilde{o}_i(t)$ . Note that to calculate the entire overall distribution,  $b(t+1)$  requires calculating each of the  $\tilde{b}_i(t+1)$ . Since  $b(t+1) = \langle \tilde{b}_1(t+1), \tilde{b}_2(t+1), \dots, \tilde{b}_k(t+1) \rangle$ , and each  $\tilde{b}_i(t+1)$  has  $n_i$  calculations, to calculate  $b(t+1)$  requires  $\sum_{i \in I} n_i = |S|$  calculations, the size of the entire state space.

The crucial part here is to consider the possible pairs of the  $s_i, s'_i$ , which are needed to calculate  $\sum_{i \in I} \sum_{s_i \in \tilde{S}_i} \mathbb{P}(s'_i|a, s_i) b_{s_i}(t)$ . Here,  $s'_i$  is used to identify  $s_i(t+1)$ , with  $s_i$  denoting  $s_i(t)$ . For a given  $a$ , if  $a$  does not affect the domain  $\tilde{S}_i$ ,  $\mathbb{P}(s'|a, s) = 0$ . Thus, it is known already that  $\sum_{i \in I} \sum_{s_i \in \tilde{S}_i} \mathbb{P}(s'_i|a, s_i) b_{s_i}(t) = 0$ . If the action  $a$  affects a single domain  $\tilde{S}_i$ , then there are  $n_i^2$  possible pairings to consider in order to update  $\tilde{b}_i$ . Further, updating the other domains unaffected by  $a$  is either  $\tilde{b}_i$  again if the observation is null, or reduced to considering  $\tilde{b}_i(t)$  and  $\tilde{o}_i(t)$ , independent of the action  $a$ . If the action  $a$  affects domains  $\tilde{S}_i$ , with  $|\tilde{S}_i| = n_i$  and  $\tilde{S}_j$ , and  $|\tilde{S}_j| = n_j$ , that is to say that given action  $a$ , there is some pair  $(s_i, s_j)$  such that  $T_a(s_i, s_j) \neq 0$  then there are at most  $n_i n_j$  pairings of  $(s_i, s_j)$  to consider, given that  $s' \in \tilde{S}_j$ .

This means that for any partitioned state space and action  $a$ , the maximum number of pairings that needs be considered is  $\sum_{i \in I} \sum_{j \in I} n_i n_j$ . This can be reduced wherever there is a domain that is unaffected by action  $a$ , say  $\tilde{S}_i, |\tilde{S}_i| = n_i$ , since  $\sum_{i \in I} \sum_{s_i \in \tilde{S}_i} \mathbb{P}(s'_i|a, s_i) b_{s_i}(t) = 0$ , a calculation of order  $n_i$ . This means that the number of pairings that need to be considered for action  $a$  is  $\sum_{i \in I} \sum_{j \in I} n_i n_j - \sum_{i \in I, T_a(\tilde{S}_i)=0} n_i$ .

Compare this to a belief distribution over the unpartitioned state space,  $S$ , where the probability of being in state  $s'$  at time  $t+1$  is calculated as:

$$b'(s') = \mathbb{P}(s'|o', a, b) = \frac{\mathbb{P}(o'|s', a, b) \mathbb{P}(s'|a, b)}{\mathbb{P}(o'|a, b)} = \frac{\mathbb{P}(o'|s', a) \sum_{s \in S} \mathbb{P}(s'|a, b, s) \mathbb{P}(s|a, b)}{\mathbb{P}(o'|a, b)} \quad (3.3)$$

$$= \frac{\mathbb{P}(o'|s', a) \sum_{s \in S} \mathbb{P}(s'|a, s) b(s)}{\mathbb{P}(o'|a, b)} \quad (3.4)$$

where  $b$  is the belief distribution at time  $t$ .  $b(s)$  denotes the probability of being in state  $s$ .  $b'$  is the updated belief distribution at time  $t+1$ ,  $a$  is the action taken at time  $t$ ,  $s$  is the unobserved state at time  $t$ ,  $s \in S$ , and  $s'$  is the unobserved state to which the machine transitions (Williams & Young, 2007, p.395). Similarly, to calculate the entire belief state  $b' = \sum_{s \in S} b'(s')$ .

If the action  $a$  affects domains  $\tilde{S}_i$ , with  $|\tilde{S}_i| = n_i$  and  $\tilde{S}_j$ , and  $|\tilde{S}_j| = n_j$ , that is to say that given action  $a$ , there is some pair  $(s_i, s_j)$  such that  $T_a(s_i, s_j) \neq 0$  then there are at most  $n_i n_j$  pairings of  $(s_i, s_j)$  to consider, given that  $s' \in \tilde{S}_j$ .

To compare the number of calculations needed in order to calculate , note that for the unpartitioned space,  $(\sum_{i \in I} n_i)^2$ :

$$\left( \sum_{i \in I} n_i \right)^2 = \sum_{i \in I} \sum_{j \in I} n_i n_j \quad (3.5)$$

At best, the unpartitioned state space is of order  $\sum_{i \in I} \sum_{j \in I} n_i n_j$ , which is at least as large as the order of the partitioned state space.

$$\sum_{i \in I} \sum_{j \in I} n_i n_j \geq \left( \sum_{i \in I} \sum_{j \in I} n_i n_j - \sum_{i \in I, T_a(\tilde{S}_i)=0} n_i \right) \quad (3.6)$$

This means, that by partitioning the state space, there are potential computational benefits, since the total number of calculations may be reduced. These computational advantages will occur whenever there is at least one action  $a$  and one domain  $\tilde{S}_i \subseteq S$  such that  $a$  has no effect on  $\tilde{S}_i$ , or when for some observational domain, there is some null observation.

Also, since each  $\tilde{b}_i$  is a distribution over  $n_i$  elements, compared with  $b$  over  $|S|$  elements, identifying deviation between  $\tilde{b}_i(t)$  and  $\tilde{b}_i(t+1)$  is easier, because there are fewer elements over which  $\tilde{b}_i$  is distributed. This means that (un)certainty of one  $\tilde{b}_i$  over domain  $\tilde{S}_i$  does not affect the (un)certainty of another  $\tilde{b}_j$  over  $\tilde{S}_j$ . As such, one  $\tilde{b}_i$  might be certain of being in a state  $s \in \tilde{S}_i$ , whilst  $\tilde{b}_j$  is uniformly distributed over  $\tilde{S}_j$ . This model can then choose an action according to this differing level of uncertainty, as the source of the uncertainty is identifiable. In this way, the model is more sensitive to the different types of observations.

# Chapter 4

## Evaluation

In order to test the claims in subsection 3.3.4, ideally a corpus of dialogue which was annotated for multiple input types would have been found. The work by Stoa et al. (2008) was the closest found, but using the gaze-tracking as an input was not replicable within the lab. As an alternative, based on the work of Shahryari & Hamilton (2016), a car advisory system was simulated. Six of the traffic signs recognised by Shahryari & Hamilton (2016) were used as states of the partitioned POMDP: Give Way, Stop, No Entry, No Left Turn, No Right Turn and Speed Limit 40. These then formed the states of the “traffic signs” partition. The other partitions were speed and navigation. In the speed class, the four states were: speed 0; speed 20; speed 40; and speed 60. The navigation class consisted of three states: Go Straight, Turn Left, and Turn Right. This resulted in a total of thirteen states for the transition, observation and reward functions.

The system was able to advise the simulated human driving the car. There were five possible advisory actions that the system could offer. These were as follows: Advise Slow Down; Advise Speed Up; Advise Turn Left; Advise Turn Right; and Advise Go Straight.

The observed behaviour of the simulated human became the observations available to the system. These six possible observations were: IncreaseSpeed; DecreaseSpeed; SetDirectionGoStraight; SetDirectionTurnLeft; SetDirectionTurnRight; NoResponse. The NoResponse observation was included to allow for those scenarios when there was no change to the system from the simulated human. The transition, observation, and reward functions were hand-crafted. The transition between speed states were limited so that only neighbouring states could be visited. I.e. from speed state 20, the system could go to speed state 0 or state 40, but not to state 60 without first going to state 40. The speed and the navigation beliefs followed on from the previous state; however the camera belief for a new traffic sign was randomly generated at the beginning of each time step.

The software was run on a 2-year-old laptop, with an Intel i7-4720 2.60GHz processor, with 16GB RAM, running 64 bit Windows 10.

## 4.1 Programming the model

The car system was written in *C#*. There were classes for each partition, plus classes for reading in the transition, observation and reward functions, and classes for performing the update function. There was also a class for classifying which update method should be used for the combination of states, according to subsection 3.3.2.

The system used a greedy policy, based on the most probable state combination. For example, if the belief vector over the speed states was  $\{0.74894, 0.19848, 0.04341, 0.00916\}$  for the states 0, 20, 40, 60 respectively, then the system would look for the rewards that matched the 0 speed state, since it has the greatest probability in this partition. Similarly, the system looked for the reward which matched the most likely traffic sign and direction. The rewards which matched all three of the most likely states were then compared for which action gave the greatest reward. This action was then the action chosen by the system.

In order to provide the comparison software with the equivalent unpartitioned POMDP, the state combinations had to be written out more explicitly. Instead of thirteen states for the partitioned model, each combination of states, one traffic sign, one speed, and one navigation, had to be described individually, for a total of 72 states. As a result, the transition function, observation function, and reward function which had been written for the partitioned POMDP had to be written explicitly for each of these 72 states. This meant that each of the 72 states had to be assigned a specific value in the functions, where before, ‘wildcard’ values had been used for all states in a partition which were not directly affected by that transition combination. This meant that converting the transition, observation and reward functions from the underspecified functions acceptable for the partitioned POMDP took additional time, although the values for the state combinations remained the same.

## 4.2 Comparison software

Software for calculating exact and approximate optimal policies include Cassandra (2003)’s POMDP-solve and Kurniawati et al. (2008)’s APPL. POMDP-solve implements exact solution algorithms to calculate the optimal policy for a given POMDP problem, when written in the POMDP file format. It is written in C, and the most recent version is 5.4, released in May 2015. It runs on GNU/Linux and versions are available for Apple OS X. POMDP-solve aims to calculate an exact solution for a problem, which as mentioned in subsection 2.2.1 is intractable for large problems. While POMDP-solve was attempted to calculate a policy for the car system, it failed after the third iteration, taking 12180.68s to calculate 2015 vectors.

As a result of this failure, APPL software was used instead, and was run on Cygwin on the above mentioned laptop. A two hour timeout limit was used, by which point 23,827 belief vectors were being considered to approximate an optimal policy. This policy was used with the model to calculate



the total expected reward after 100 time steps. The time taken to simulate 100 timesteps was then measured and compared against the partitioned car model software.

### 4.3 Results

As the key theoretical result described in subsection 3.3.4 was about the time improvements, the time taken to simulate 100 timesteps 1000 times was measured, for both the APPL software and for the partitioned Car Model system. This 1000 simulations of 100 timesteps was repeated 10 times for each of the APPL software and for the Partitioned Model. The results in seconds are:

APPL	338.23	345.60	333.04	332.96	334.18	339.40	335.74	335.79	337.57	342.38
Partitioned	91.515	92.300	95.272	93.154	94.050	94.438	93.247	94.296	93.661	95.713

The average for the APPL software was 337.489 seconds, with standard deviation of 4.09. The average for the partitioned software was 92.802 with standard deviation of 1.61. This means that the partitioned model takes just 27.5% of the time that the unpartitioned model does. This can partly be explained by how the partitioned model treats the NoResponse observation, which occurs approximately one third of the time. The unpartitioned model treats this just as any other observation, and updates according to the POMDP update method. However, the partitioned model will update the system according to either method 1. or method 4. as defined in subsection 3.3.2, since the received observation does not affect the speed and direction partitions directly. Whether the update is method 1. or method 4 will be dependent on which partition, and whether the last action from the system was relevant or not. For example, if the last advice was to slow down, and a NoResponse observation was received, then the direction would update according to method 1. and the speed would update according to method 4. This accounts for a speed increase of up to one third. The reason for the remaining increase in speed can be attributed to what occurs for the remaining observations. The observations IncreaseSpeed and DecreaseSpeed affect only the speed, and are independent of the direction and of the camera. Similarly, the observations SetDirectionGoStraight, SetDirectionTurnLeft, and SetDirectionTurnRight affect only the direction and not the speed or the camera. As such, for any of these observations, only one of the speed or navigation partitions is updated according to the usual POMDP update function of method 3. The other will update according to method 2. or method 1. depending on whether the last advice was relevant or not. Thus, for the remaining observations, only half of them will need to be updated according to the usual POMDP methods. This accounts for most of the remaining time improvements.

## 4.4 Discussion

A key point mentioned in section 4.1 is that to convert the transition, observation, and reward functions from the partitioned POMDP to the unpartitioned POMDP format, takes time and requires each of the 72 states to be explicitly allocated a probability. This is because the partitioned POMDP allows for underspecification when one partition is not relevant to the update function. For example, the transition probability for the state Give Way - 20 - GoStraight, with action AdviseSlowDown, mimics the transition probability for the state Give Way - 20 - TurnLeft with action AdviseSlowDown, and likewise with the state Give Way - 20 - TurnRight with action AdviseSlowDown. In each of these, the direction is unchanged in the following state, as it is independent of this particular action. What is important to this particular action of AdviseSlowDown is the traffic sign and the current speed. As such, the transition probabilities for the partitioned POMDP can be written as  $\mathbb{P}(s'|a = AdviseSlowDown, s_T = GiveWay, s_S = 20)$ , independent of all navigation options  $s_N \in \{GoStraight, TurnLeft, TurnRight\}$ , where  $s'$  is the next state combination. This contrasts with the POMDP format for the APPL software, where each probability must be explicitly defined. In this particular example, one line is written instead of three. When extended over each of the entire transition function, for any given traffic sign, either the speed partition is irrelevant or the navigation partition is irrelevant (but not both at the same time). With three navigation states and four speed states, this reduces the number of lines in the transition function by at least one third.

The time differences in the simulations is broadly in line with the predictions made in subsection 3.3.4. Where an observation is not relevant to update a partition, speed or direction, that partition updates via a different method - method 1. or method 4. Similarly, when the action is not relevant to a partition, the updates happen via a different method - method 1 or method 2. While extra input data is necessary in order to classify which update method should be used, this is countered by fewer total lines within the transition, reward and observation functions.

This particular model was ideal for partitioning, since speed, navigation and traffic signs can reasonably assumed to be all independent. Further, the actions, and the observations, only affected a single partition at a time, allowing the remaining partitions to be updated by simpler methods. Such dramatic improvements are not to be expected for every implementation of a partitioned model; the theoretical results only guarantee an improvement when an action does not affect at least one partition.

## Chapter 5

# Conclusion

This study set out to investigate dialogue managers within the context of a dialogue system, and to identify how the mathematical model used for dialogue managers behaves.

This report has described the field of spoken dialogue systems and human-computer interaction. It has reviewed the mathematics and literature of Partially Observable Markov Decision Processes (POMDPs), and their application and use in dialogue managers. It has reviewed the literature of solving and approximating policies for POMDPs. It has described work on training a POMDP model from the HCRC maptask corpus, and my theoretical model to partition the POMDP state space, together with theoretical results on time improvements. These theoretical findings are a key result, as they demonstrate under what conditions a partitioned POMDP is of benefit. Note that this theoretical model is a generalised version of a POMDP, since a POMDP is simply a partitioned POMDP where all states are handled within a single partition.

The research has been limited by the lack of available data to implement the partitioned POMDP within the setting of a multimodal dialogue system, to demonstrate the relevance of the model to the field which inspired it. However, the simulated data on the Car Advisory system supports the theoretical results of a time improvement which is directly related to the independence of actions and partitions. A key constraint is that, due to the novelty of the model, software to calculate or approximate an optimal policy for it does not yet exist. An optimal policy for each of the separate partitions would not combine to an optimal policy for the whole. Further work will be needed to overcome this limitation. However, due to the multiple belief distributions, the model may be well-suited to multivariate analysis. Another limitation is that it is extremely difficult to prove that the partitioned model may be more sensitive to changes in the belief states. The APPL software does not output the belief vectors which it calculates at each time step of a simulation, which makes it impossible to compare it with the belief vectors of the partitioned software. Thus, the study can only demonstrate the time improvements from the theoretical results.

## 5.1 Future Work

Comparing policies for the two models would be challenging. Each policy is created for the individual model, and depends on its particular inputs and structure. The optimal policy approximated by Sarsop has a list of 1282 vectors of length 72 (the total number of states) for a given action and given observation, after a timeout of 2 hours. Some action-observation pairs are repeated. The policy for the partitioned POMDP was not optimised, but was a greedy policy. That is to say, the action taken was whichever action that would provide the highest immediate reward. This is because there is no available software for calculating the partitioned POMDP model policy, due to its novelty. For these reasons, comparing the policies directly would be ineffective in evaluating the system.

Ideally, a method of analysis would be to compare the belief state of each model at each timestep. While this would be straightforward with my own model, this also requires that the comparison software, APPL (Kurniawati et al., 2008), be edited to output the belief state at each time step. I attempted to do this. I identified the `nextBelSt` object in the code at Line 393, `SimulationEngine.cpp`, which contains the belief as a `SparseVector` object. I attempted to use `cout << nextBelState -i bvec` to output the belief. However, this is a pointer to a `SparseVector` object, which currently is displayed as an address. In order to display this as a dense vector, I looked at the `SparseVector` code, to see how the vectors are being stored. It appears to have been written by the original programmers, rather than using a library designed to handle sparse vectors. The documentation for this particular `SparseVector` library does not describe what method is being used to store the sparse vectors efficiently. One library that is available for handling sparse vectors is `uBLAS`, available from Boost, which provides a set of libraries for the C++ language that are peer-reviewed. This library could handle much of the functionality that APPL requires, and has documentation readily available for it. Editing the partitioned POMDP software to output the beliefs over each of the partitions would be straightforward, as they are stored in arrays.

The next step would be converting the beliefs over the partitions to a single vector which could be compared with the belief from the APPL software. This would be done in the following manner: For the timestep  $t$ , for each partition  $S_i$ , for each component  $j \in J$  such that  $s_j \in S_i$ , calculate  $\prod_i s_j$ . So for the three partitions of the car advisory model, this would be the belief value of each component of traffic signs partition, each component of the speed partition, and each component of the navigation partition; multiplied together. This would give a total of 72 outputs, equivalent to the dimension of the belief state given from the APPL software. These can then be directly compared. This would allow any discrepancies between the belief states to be identified. Thus, the quality of the partitioned POMDP model could be evaluated, not only where the belief states differ, but where this causes different actions to be taken.

## 5.2 Final Remarks

This study has shown that partitioning a POMDP model can improve the time taken to simulate a POMDP, making it three times faster in this particular scenario. In general however, such substantial improvements cannot be guaranteed; only that improvements will occur whenever an action occurs independently of a partition. The time improvements will correlate directly to how often such an action occurs, and to the size of partition which is independent of said action. This result, with its associated constraints, is significant as it will allow some large POMDPs to be reduced in size via partitioning, whenever independent partitions can be identified. This will make POMDPs more tractable when dealing with multiple input types. It would be interesting to see the model applied to a multimodal dialogue system, acting as the dialogue manager. Inputs such as automatic speech recognition, and associated prosodic features, would be of particular interest. A model trained on a corpus of sarcastic voices, with the above inputs would be worthwhile, as this model may have a potential use to detect sarcasm.

This research has identified an area where this model could have significant impact. By developing a mathematical model which is designed for multiple input types via its partitions, this research is ideally placed to provide a new type of dialogue manager for any multimodal dialogue system.

# List of Figures

- 1.1 Diagram of a SDS . . . . . 2
- 2.1 Diagram of a Markov Decision Process . . . . . 5
- 2.2 Diagram of a POMDP . . . . . 9
- 3.1 Diagram of a SDS . . . . . 19

## Bibliography

- Åström, K. J. (1965). Optimal control of markov processes with incomplete state information. *Journal of Mathematical Analysis and Applications*, 10(1), 174–205.
- Blei, D. M., Ng, A. Y., & Jordan, M. I. (2003). Latent dirichlet allocation. *the Journal of machine Learning research*, 3, 993–1022.
- Cao, X.-R., & Guo, X. (2004). Partially Observable Markov Decision Processes with reward information. In *Decision and control, 2004. cdc. 43rd ieee conference on* (Vol. 4, pp. 4393–4398).
- Cassandra, A. R. (2003). *Pomdp solve*. Retrieved from <http://www.pomdp.org/code/index.html> (Accessed 19-09-2017)
- Chadès, I., McDonald-Madden, E., McCarthy, M. A., Wintle, B., Linkie, M., & Possingham, H. P. (2008). When to stop managing or surveying cryptic threatened species. *Proceedings of the National Academy of Sciences*, 105(37), 13936–13940. Retrieved from <http://www.pnas.org/content/105/37/13936> doi: 10.1073/pnas.0805265105
- Chinaei, H. R., & Chaib-draa, B. (2011). Learning dialogue POMDP models from data. In *Advances in artificial intelligence* (pp. 86–91). Springer.
- Chinaei, H. R., Chaib-draa, B., & Lamontagne, L. (2009). Learning User Intentions in Spoken Dialogue Systems. In *Icaart* (pp. 107–114).
- Clark, H. H. (1996). *Using language*. Cambridge University Press.
- Garey, M. R., & Johnson, D. S. (1979). *Computers and intractability: A guide to the theory of np-completeness*. New York, NY, USA: W. H. Freeman & Co.
- Grice, H. P. (1989). *Studies in the Way of Words*. Harvard University Press.
- Gruber, A., Weiss, Y., & Rosen-Zvi, M. (2007). Hidden Topic Markov models. In *International conference on artificial intelligence and statistics* (pp. 163–170).
- Hauskrecht, M., & Fraser, H. (2000). Planning treatment of ischemic heart disease with partially observable markov decision processes. *Artificial Intelligence in Medicine*, 18(3), 221 - 244. Retrieved from <http://www.sciencedirect.com/science/article/pii/S09333365799000421> doi: [https://doi.org/10.1016/S0933-3657\(99\)00042-1](https://doi.org/10.1016/S0933-3657(99)00042-1)
- Hsiao, K., Kaelbling, L. P., & Lozano-Perez, T. (2007). Grasping POMDPs. In *Robotics and automation, 2007 ieee international conference on* (pp. 4685–4692).
- Jurafsky, D., Martin, J. H., Kehler, A., Vander Linden, K., & Ward, N. (2000). *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition* (Vol. 2). MIT Press.

- Kaelbling, L. P., Littman, M. L., & Cassandra, A. R. (1998). Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1), 99–134.
- Kurniawati, H., Hsu, D., & Lee, W. S. (2008). SARSOP: Efficient Point-Based POMDP Planning by Approximating Optimally Reachable Belief Spaces. In *Robotics: Science and systems* (Vol. 2008).
- Lison, P. (2015). A hybrid approach to dialogue management based on probabilistic rules. *Computer Speech & Language*, 34(1), 232–255.
- Littman, M. L., Cassandra, A. R., & Kaelbling, L. P. (1995). Efficient Dynamic-programming Updates in Partially Observable Markov Decision Processes.
- Papadimitriou, C. H., & Tsitsiklis, J. N. (1987). The complexity of Markov decision processes. *Mathematics of operations research*, 12(3), 441–450.
- Pineau, J., Gordon, G., & Thrun, S. (2003). Point-based value iteration: An anytime algorithm for POMDPs. In *Ijcai* (Vol. 3, pp. 1025–1032).
- Schröder, M. (2010, January). The SEMAINE API: Towards a Standards-based Framework for Building Emotion-oriented Systems. *Adv. in Hum.-Comp. Int., 2010*. Retrieved from <http://dx.doi.org/10.1155/2010/319406> doi: 10.1155/2010/319406
- Shahryari, S., & Hamilton, C. (2016). Neural network-pomdp-based traffic sign classification under weather conditions. In R. Khoury & C. Drummond (Eds.), *Advances in artificial intelligence: 29th canadian conference on artificial intelligence, canadian ai 2016, victoria, bc, canada, may 31 - june 3, 2016. proceedings* (pp. 122–127). Cham: Springer International Publishing. Retrieved from [http://dx.doi.org/10.1007/978-3-319-34111-8\\_17](http://dx.doi.org/10.1007/978-3-319-34111-8_17) doi: 10.1007/978-3-319-34111-8\_17
- Stoia, L., Shockley, D. M., Byron, D. K., & Fosler-Lussier, E. (2008, May). Scare: A situated corpus with annotated referring expressions. In *Proceedings of the 6th international conference on language resources and evaluation (lrec 2008)*. Marrakesh, Morocco.
- Temizer, S., Kochenderfer, M., Kaelbling, L., Lozano-Pérez, T., & Kuchar, J. (2010). Collision avoidance for unmanned aircraft using markov decision processes. In *Aiaa guidance, navigation, and control conference* (p. 8040).
- Thomson, B., Schatzmann, J., Weilhammer, K., Ye, H., & Young, S. (2007). Training a real-world POMDP-based Dialogue System. In *Proceedings of the workshop on bridging the gap: Academic and industrial research in dialog technologies* (pp. 9–16).
- Varges, S., Riccardi, G., Quarteroni, S., & Ivanov, A. (2011, May). POMDP concept policies and task structures for hybrid dialog management. In *Acoustics, speech and signal processing (icassp), 2011 ieee international conference* (p. 5592–5595). doi: 10.1109/ICASSP.2011.5947627



- White, D. J. D. J. (1993). *Markov Decision Processes* [Book; Book/Illustrated]. Chichester [England] ; New York : John Wiley & Sons.
- Williams, J. D., & Young, S. (2007). Partially observable Markov decision processes for spoken dialog systems. *Computer Speech & Language*, *21*(2), 393–422.
- Wu, G., Yuan, C., Leng, B., & Wang, X. (2015). Finite-to-infinite n-best pomdp for spoken dialogue management. In *Chinese computational linguistics and natural language processing based on naturally annotated big data* (pp. 369–380). Springer.
- Yoshino, K., & Kawahara, T. (2014). Information navigation system based on POMDP that tracks user focus. In *15th annual meeting of the special interest group on discourse and dialogue* (p. 32).
- Young, S. (2010, May). Cognitive User Interfaces. *Signal Processing Magazine, IEEE*, *27*(3), 128-140. doi: 10.1109/MSP.2010.935874
- Young, S., Gašić, M., Keizer, S., Mairesse, F., Schatzmann, J., Thomson, B., & Yu, K. (2010). The Hidden Information State Model: A practical framework for POMDP-based spoken dialogue management. *Computer Speech & Language*, *24*(2), 150–174.
- Young, S., Gasic, M., Thomson, B., & Williams, J. (2013, May). POMDP-Based Statistical Spoken Dialog Systems: A Review. *Proceedings of the IEEE*, *101*(5), 1160-1179. doi: 10.1109/JPROC.2012.2225812

# Appendix

Documentation of Car Advisory Model

# **Car Advisory System**

Grace Cowderoy  
9/26/2017 2:26:00 PM



# Table of Contents

Namespace Index .....	2
Class Index .....	3
AI .....	4
Class Documentation .....	5
AI.BeliefUpdater .....	5
Camera .....	7
Navigation .....	9
AI.Observation .....	12
AI.Program .....	13
AI.RewardStateAction .....	14
Speed .....	15
AI.Transition .....	17
AI.UpdateClassifier .....	18
Index .....	19



# Namespace Index

## Packages

Here are the packages with brief descriptions (if available):

<b>AI</b> .....	4
-----------------	---

# Class Index

## Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

<b>AI.BeliefUpdater (Step process to update the belief state. When given the action and observation, this class holds the observation probability for the state. It also holds the summed transition probabilities for the state-action-next state combination. )</b>	5
<b>Camera (Camera sensor to provide visual information to the car's AI system. Provides a vector of the possible traffic signs that the camera believes it is seeing and the probability that the object is being seen correctly. )</b>	7
<b>Navigation (Allows the retrieval of current and planned direction actions as a belief vector. Options are: "Go Straight", "Turn Left", and "Turn Right" )</b>	9
<b>AI.Observation (Observation Class for reading in the observation function from a CSV file. )</b>	12
<b>AI.Program ()</b>	13
<b>AI.RewardStateAction (Object for reading the Reward function. Each line in the csv file is read into a RewardStateAction object. This is of the form: traffic sign, speed, direction, advised action, reward given. Program then maintains a list of RewardStateAction objects. )</b>	14
<b>Speed (Allows the retrieval of the car's speed as a belief vector. Also allows setting the car's speed. )</b>	15
<b>AI.Transition (Class for the Transition function. Each line in the csv file is read into Transition object. Format is: last advice from system, current speed, current direction, next speed, next direction, probability. Program.cs will maintain a list of Transition objects. )</b>	17
<b>AI.UpdateClassifier (This class is to hold how each combination of Action and Observation for Speed and Navigation should be updated. Each combination is given a number from 1-4. Depending on this number, Program.cs updates the speed or the navigation according to its switch statement. See #region DirectionUpdater in Program.cs )</b>	18



# Namespace Documentation

## AI Namespace Reference

### Classes

- class **BeliefUpdater**
- *Step process to update the belief state. When given the action and observation, this class holds the observation probability for the state. It also holds the summed transition probabilities for the state-action-next state combination.* class **Observation**
- **Observation** *Class for reading in the observation function from a CSV file.* class **Program**
- class **RewardStateAction**
- *Object for reading the Reward function. Each line in the csv file is read into a **RewardStateAction** object. This is of the form: traffic sign, speed, direction, advised action, reward given.* **Program** then maintains a list of **RewardStateAction** objects. class **Transition**
- *Class for the **Transition** function. Each line in the csv file is read into **Transition** object. Format is: last advice from system, current speed, current direction, next speed, next direction, probability. Program.cs will maintain a list of **Transition** objects.* class **UpdateClassifier**  
*This class is to hold how each combination of Action and **Observation** for **Speed** and **Navigation** should be updated. Each combination is given a number from 1-4. Depending on this number, Program.cs updates the speed or the navigation according to its switch statement. See #region DirectionUpdater in Program.cs*

# Class Documentation

## AI.BeliefUpdater Class Reference

Step process to update the belief state. When given the action and observation, this class holds the observation probability for the state. It also holds the summed transition probabilities for the state-action-next state combination.

### Public Member Functions

- **BeliefUpdater** (string identifier, float obsProb, float transProbSum)  
*Constructor for BeliefUpdater. This is used as a place holder while updating the speed and navigation belief vectors.*
- string **GetIdentifier** ()  
*To retrieve the identifier from a **BeliefUpdater** object.*
- float **GetObsProb** ()  
*To retrieve the observation probability from a **BeliefUpdater** object.*
- float **GetTransProbSum** ()  
*To retrieve the summation of the relevant transition probabilities from a **BeliefUpdater** object.*

---

### Detailed Description

Step process to update the belief state. When given the action and observation, this class holds the observation probability for the state. It also holds the summed transition probabilities for the state-action-next state combination.

---

### Constructor & Destructor Documentation

#### AI.BeliefUpdater.BeliefUpdater (string identifier, float obsProb, float transProbSum)

Constructor for BeliefUpdater. This is used as a place holder while updating the speed and navigation belief vectors.

#### Parameters:

<i>identifier</i>	Speed or Navigation state
<i>obsProb</i>	Probability of last observation, given the state and the last advice
<i>transProbSum</i>	Summation of the transition probabilities for the state and last advice

---

### Member Function Documentation

#### string AI.BeliefUpdater.GetIdentifier ()

To retrieve the identifier from a **BeliefUpdater** object.

**Returns:**

identifier

**float AI.BeliefUpdater.GetObsProb ()**

To retrieve the observation probability from a **BeliefUpdater** object.

**Returns:**

obsProb

**float AI.BeliefUpdater.GetTransProbSum ()**

To retrieve the summation of the relevant transition probabilities from a **BeliefUpdater** object.

**Returns:**

transProbSum

---

**The documentation for this class was generated from the following file:**

- C:/Users/Random/Documents/TCDWork/UMLModel/GraceModelCSVRead/Grace/BeliefUpdater.cs

## Camera Class Reference

**Camera** sensor to provide visual information to the car's **AI** system. Provides a vector of the possible traffic signs that the camera believes it is seeing and the probability that the object is being seen correctly.

### Public Member Functions

- **Camera ()**  
*Constructor for the **Camera** class.*
- void **UpdateTrafficSigns ()**  
*Method to update **Camera** belief vector with new probabilities.*
- string **GetBeliefObject ()**  
*Method to get the most likely traffic sign.*
- float **GetBeliefProbability ()**  
*Method to get the highest belief probability.*
- ArrayList **GetPossibleTrafficSigns ()**  
*List of possible traffic signs*
- Dictionary< string, float > **GetTrafficSignsVector ()**  
*Method to return the belief vector of the various traffic signs.*
- void **addOrUpdateDict** (Dictionary< string, float > dic, string key, float newValue)  
*Method to update the camera class belief vector.*

---

### Detailed Description

**Camera** sensor to provide visual information to the car's **AI** system. Provides a vector of the possible traffic signs that the camera believes it is seeing and the probability that the object is being seen correctly.

---

### Constructor & Destructor Documentation

#### **Camera.Camera ()**

Constructor for the **Camera** class.

---

### Member Function Documentation

**void Camera.addOrUpdateDict** (Dictionary< string, float > *dic*, string *key*, float *newValue*)

Method to update the camera class belief vector.

#### **Parameters:**

<i>dic</i>	Dictionary to be updated
------------	--------------------------

<i>key</i>	Key in dic to be updated
<i>newValue</i>	New value to be assigned to key

### **string Camera.GetBeliefObject ()**

Method to get the most likely traffic sign.

#### **Returns:**

Returns the beliefObject variable

### **float Camera.GetBeliefProbability ()**

Method to get the highest belief probability.

#### **Returns:**

Returns the beliefProbability variable

### **ArrayList Camera.GetPossibleTrafficSigns ()**

List of possible traffic signs

#### **Returns:**

Returns vector of traffic signs

### **Dictionary<string, float> Camera.GetTrafficSignsVector ()**

Method to return the belief vector of the various traffic signs.

#### **Returns:**

Returns dictionary of traffic signs.

### **void Camera.UpdateTrafficSigns ()**

Method to update **Camera** belief vector with new probabilities.

#### **Returns:**

Returns the updated traffic signs vector.

---

**The documentation for this class was generated from the following file:**

- C:/Users/Random/Documents/TCDWork/UMLModel/GraceModelCSVRead/Grace/Camera.cs

## Navigation Class Reference

Allows the retrieval of current and planned direction actions as a belief vector. Options are: "Go Straight", "Turn Left", and "Turn Right"

### Public Member Functions

- **Navigation ()**  
*Constructor for the navigation class. Has the directions "Go Straight", "Turn Left", and "Turn Right". A random number generator then allocates probabilities to the initial object.*
- **ArrayList GetPossibleNavigation ()**  
*Method to get the list of possible directions*
- **Dictionary< string, float > GetNavigationVector ()**  
*Method to get the possible directions with their assigned probabilities.*
- **string GetCurrentAction ()**
- **string GetPlannedAction ()**
- **void SetPlannedAction (string plannedAction)**
- **void AddOrUpdateDict (Dictionary< string, float > dic, string key, float newValue)**  
*Method to update the current or plannedDirection vector, to add a direction in.*
- **void SetPlannedNavigationVector (string key, float newValue)**  
*Method to set the planned navigation vector.*
- **Dictionary< string, float > GetPlannedNavigationVector ()**  
*Method to retrieve the plannedNavigation vector*
- **void SetPlannedNavigation (Dictionary< string, float > updatedNavigationVector)**  
*Method to update the navigation vector with a set dictionary.*
- **void PlannedToCurrentNavVector ()**  
*Method to replace the currentNavigation vector with the planned navigation vector.*

---

### Detailed Description

Allows the retrieval of current and planned direction actions as a belief vector. Options are: "Go Straight", "Turn Left", and "Turn Right"

---

### Constructor & Destructor Documentation

#### Navigation.Navigation ()

Constructor for the navigation class. Has the directions "Go Straight", "Turn Left", and "Turn Right". A random number generator then allocates probabilities to the initial object.

---

### Member Function Documentation

**void Navigation.AddOrUpdateDict (Dictionary< string, float > dic, string key, float newValue)**

Method to update the current or plannedDirection vector, to add a direction in.

**Parameters:**

<i>dic</i>	Dictionary to be updated
<i>key</i>	Key to be updated
<i>newValue</i>	New value to be assigned to key.

**string Navigation.GetCurrentAction ()**

Returns currentAction variable

**Dictionary<string, float> Navigation.GetNavigationVector ()**

Method to get the possible directions with their assigned probabilities.

**Returns:**

Returns a dictionary of directions with their currently assigned probabilities

**string Navigation.GetPlannedAction ()**

Returns plannedAction variable

**Dictionary<string, float> Navigation.GetPlannedNavigationVector ()**

Method to retrieve the plannedNavigation vector

**Returns:**

Returns the Planned **Navigation** vector

**ArrayList Navigation.GetPossibleNavigation ()**

Method to get the list of possible directions

**Returns:**

Returns a list of possible directions

**void Navigation.PlannedToCurrentNavVector ()**

Method to replace the currentNavigation vector with the planned navigation vector.

**void Navigation.SetPlannedAction (string *plannedAction*)**

Sets the plannedAction variable

**void Navigation.SetPlannedNavigation (Dictionary< string, float > *updatedNavigationVector*)**

Method to update the navigation vector with a set dictionary.

**Parameters:**

<i>updatedNavigationVector</i>	Dictionary to be set as the planned navigation vector.
--------------------------------	--

**void Navigation.SetPlannedNavigationVector (string *key*, float *newValue*)**

Method to set the planned navigation vector.

**Parameters:**

<i>key</i>	Key to be edited in plannedNavigationVector
<i>newValue</i>	New value to be assigned to key.

---

**The documentation for this class was generated from the following file:**

- C:/Users/Random/Documents/TCDWork/UMLModel/GraceModelCSVRead/Grace/Navigation.cs



## AI.Observation Class Reference

**Observation** Class for reading in the observation function from a CSV file.

### Public Member Functions

- **Observation** (string trafficSign, string advice, string observation, float probability)  
*Object for reading in the **Observation** function - a traffic sign, an advised action, an observation, and the probability. Each line in the CSV file creates an **Observation** object. Program.cs then maintains a list of **Observation** objects.*

### Public Attributes

- readonly string **observation**
  - readonly string **trafficSign**
  - readonly string **advice**
  - readonly float **probability**
- 

### Detailed Description

**Observation** Class for reading in the observation function from a CSV file.

---

### Constructor & Destructor Documentation

**AI.Observation.Observation** (string *trafficSign*, string *advice*, string *observation*, float *probability*)

Object for reading in the **Observation** function - a traffic sign, an advised action, an observation, and the probability. Each line in the CSV file creates an **Observation** object. Program.cs then maintains a list of **Observation** objects.

#### Parameters:

<i>trafficSign</i>	Last traffic sign state
<i>advice</i>	Last advice given by the system
<i>observation</i>	Last observation received by the system
<i>probability</i>	Probability of receiving observation given the advice and last traffic sign state.

---

The documentation for this class was generated from the following file:

- C:/Users/Random/Documents/TCDWork/UMLModel/GraceModelCSVRead/Grace/Observation.cs

## AI.Program Class Reference

### Static Public Member Functions

- static ArrayList **actionList** ()  
*Defining the possible advisory actions that are available to the system. list type="bullet"> item>Advise **Speed** Up item>Advise Slow Down item>Advise Turn Left item>Advise Turn Right item>Advise Go Straight /list> /summary>*
- 

### Detailed Description

Partitioned POMDP simulation on a car advisory system. There are three partitions: one for traffic signs; one for speed; and one for direction. The traffic sign partition contains the signs: Give Way; Stop; No Entry; No Left Turn; No Right Turn; and **Speed** Limit 40. The speed partition contains integer speed states, at 0, 20, 40 and 60. The direction partition contains the states: Go Straight; Turn Left; and Turn Right.

The actions available to the system are advisories: Advise **Speed** Up; Advise Slow Down; Advise Turn Left; Advise Turn Right; and Advise Go Straight.

The actor or user may respond with: SetSpeed(Increase); SetSpeed(Decrease); SetDirection(Turn Left); SetDirection(Turn Right); SetDirection(Go Straight).

The camera object generates a random vector over all possible traffic signs - the belief distribution that it has identified a given traffic sign. The speed object generates a random speed between 0 and 70. The object then returns a distribution over the possible speed states. This distribution is a distance measure from the two nearest speed states. The navigation objects generates a random vector over the three possible directions - as a belief distribution over in which direction the car is travelling.

#### Returns:

---

The documentation for this class was generated from the following file:

- C:/Users/Random/Documents/TCDWork/UMLModel/GraceModelCSVRead/Grace/Program.cs

## AI.RewardStateAction Class Reference

Object for reading the Reward function. Each line in the csv file is read into a **RewardStateAction** object. This is of the form: traffic sign, speed, direction, advised action, reward given. **Program** then maintains a list of **RewardStateAction** objects.

### Public Member Functions

- **RewardStateAction** (string trafficSign, int speed, string direction, string advisedAction, int reward)

### Public Attributes

- readonly string **trafficSign**
  - readonly int **speed**
  - readonly string **direction**
  - readonly string **advisedAction**
  - readonly int **reward**
- 

### Detailed Description

Object for reading the Reward function. Each line in the csv file is read into a **RewardStateAction** object. This is of the form: traffic sign, speed, direction, advised action, reward given. **Program** then maintains a list of **RewardStateAction** objects.

---

### Constructor & Destructor Documentation

**AI.RewardStateAction.RewardStateAction** (string *trafficSign*, int *speed*, string *direction*, string *advisedAction*, int *reward*)

#### Parameters:

<i>trafficSign</i>	Most likely traffic sign
<i>speed</i>	Most likely speed state
<i>direction</i>	Most likely direction state
<i>advisedAction</i>	Advised action for combination of traffic sign, speed and direction states.
<i>reward</i>	Reward associated with state - action pairing.

---

The documentation for this class was generated from the following file:

- C:/Users/Random/Documents/TCDWork/UMLModel/GraceModelCSVRead/Grace/RewardStateAction.cs

## Speed Class Reference

Allows the retrieval of the car's speed as a belief vector. Also allows setting the car's speed.

### Public Member Functions

- **Speed ()**  
*Allows the retrieval and setting of the car's speed.*
- **int GetSpeed ()**
- **ArrayList GetSpeedStates ()**
- **void SetSpeed (int speed)**  
*Setting the actual speed.*
- **void SetSpeed (string setspeed)**  
*Method to change the actual speed from the strings "Increase" or "Decrease"*
- **Dictionary< int, float > InitialiseSpeedProbabilities ()**
- **Dictionary< int, float > GetCurrentSpeedProbabilities ()**  
*Method to retrieve the current speed probabilities.*
- **void SetUpdatedSpeedVector (Dictionary< int, float > updatedSpeedVector)**  
*Method to update the speed vector.*
- **void UpdatedToCurrentSpeedVector ()**  
*Sets the updatedSpeedVector as the new current speed vector.*

---

### Detailed Description

Allows the retrieval of the car's speed as a belief vector. Also allows setting the car's speed.

---

### Constructor & Destructor Documentation

#### Speed.Speed ()

Allows the retrieval and setting of the car's speed.

---

### Member Function Documentation

#### Dictionary<int, float> Speed.GetCurrentSpeedProbabilities ()

Method to retrieve the current speed probabilities.

#### Returns:

Returns currentSpeedProbabilities

#### int Speed.GetSpeed ()

Returns the speed variable

### **ArrayList Speed.GetSpeedStates ()**

accesses the speedStates

### **Dictionary<int, float> Speed.InitialiseSpeedProbabilities ()**

- This uses a distance measure from actual speed to the speed states.
- This assumes that the gaps between the speed states are all 20.
- Looks for the closest two states, and assigns a probability depending on the distance, up to a maximum of 0.85.
- If the distance is greater than 20, then a tail of either 0.1 or 0.05 is applied.
- This is to make updating the probabilities easier.

**Returns:**

### **void Speed.SetSpeed (int speed)**

Setting the actual speed.

**Parameters:**

<i>speed</i>	New speed int value.
--------------	----------------------

### **void Speed.SetSpeed (string setspeed)**

Method to change the actual speed from the strings "Increase" or "Decrease"

**Parameters:**

<i>setspeed</i>	String input to increase or decrease speed. Use "Increase" or "Decrease".
-----------------	---

### **void Speed.SetUpdatedSpeedVector (Dictionary< int, float > updatedSpeedVector)**

Method to update the speed vector.

**Parameters:**

<i>updatedSpeedVect</i> or <i>or</i>	The new speed vector to set as the updated speed vector.
--	--

### **void Speed.UpdatedToCurrentSpeedVector ()**

Sets the updatedSpeedVector as the new current speed vector.

---

**The documentation for this class was generated from the following file:**

- C:/Users/Random/Documents/TCDWork/UMLModel/GraceModelCSVRead/Grace/Speed.cs

## AI.Transition Class Reference

Class for the **Transition** function. Each line in the csv file is read into **Transition** object. Format is: last advice from system, current speed, current direction, next speed, next direction, probability. Program.cs will maintain a list of **Transition** objects.

### Public Member Functions

- **Transition** (string lastAdvice, int currentSpeed, string currentNavigation, int nextSpeed, string nextNavigation, float probability)  
*Class to read in the transition function.*

### Public Attributes

- readonly string **lastAdvice**
- readonly int **currentSpeed**
- readonly string **currentNavigation**
- readonly int **nextSpeed**
- readonly string **nextNavigation**
- readonly float **probability**

---

### Detailed Description

Class for the **Transition** function. Each line in the csv file is read into **Transition** object. Format is: last advice from system, current speed, current direction, next speed, next direction, probability. Program.cs will maintain a list of **Transition** objects.

---

### Constructor & Destructor Documentation

**AI.Transition.Transition** (string *lastAdvice*, int *currentSpeed*, string *currentNavigation*, int *nextSpeed*, string *nextNavigation*, float *probability*)

Class to read in the transition function.

#### Parameters:

<i>lastAdvice</i>	Last advice given by the system.
<i>currentSpeed</i>	Most recent speed state.
<i>currentNavigation</i>	Most recent navigation state.
<i>nextSpeed</i>	Possible next speed state.
<i>nextNavigation</i>	Possible next navigation state.
<i>probability</i>	Probability of going from current speed/navigation states to nextSpeed and nextNavigation states.

---

The documentation for this class was generated from the following file:

- C:/Users/Random/Documents/TCDWork/UMLModel/GraceModelCSVRead/Grace/Transition.cs

## AI.UpdateClassifier Class Reference

This class is to hold how each combination of Action and **Observation** for **Speed** and **Navigation** should be updated. Each combination is given a number from 1-4. Depending on this number, Program.cs updates the speed or the navigation according to its switch statement. See #region DirectionUpdater in Program.cs

### Public Member Functions

- **UpdateClassifier** (string advice, string observation, int updateClass)  
*Constructor for the UpdateClassifier object. This is a place holder for the update classifier functions.*

### Public Attributes

- readonly string **advice**
- readonly string **observation**
- readonly int **updateClass**

---

### Detailed Description

This class is to hold how each combination of Action and **Observation** for **Speed** and **Navigation** should be updated. Each combination is given a number from 1-4. Depending on this number, Program.cs updates the speed or the navigation according to its switch statement. See #region DirectionUpdater in Program.cs

---

### Constructor & Destructor Documentation

#### AI.UpdateClassifier.UpdateClassifier (string *advice*, string *observation*, int *updateClass*)

Constructor for the **UpdateClassifier** object. This is a place holder for the update classifier functions.

#### Parameters:

<i>advice</i>	Last advice received
<i>observation</i>	Last observation received
<i>updateClass</i>	Class in CaseUpdater CSV file for which update method will be used.

---

#### The documentation for this class was generated from the following file:

- C:/Users/Random/Documents/TCDWork/UMLModel/GraceModelCSVRead/Grace/UpdateClassifier.cs

# Index

- addOrUpdateDict
  - Camera, 7
- AddOrUpdateDict
  - Navigation, 9
- AI, 4
- AI.BeliefUpdater, 5
- AI.Observation, 12
- AI.Program, 13
- AI.RewardStateAction, 14
- AI.Transition, 17
- AI.UpdateClassifier, 18
- AI::BeliefUpdater
  - BeliefUpdater, 5
  - GetIdentifier, 5
  - GetObsProb, 6
  - GetTransProbSum, 6
- AI::Observation
  - Observation, 12
- AI::RewardStateAction
  - RewardStateAction, 14
- AI::Transition
  - Transition, 17
- AI::UpdateClassifier
  - UpdateClassifier, 18
- BeliefUpdater
  - AI::BeliefUpdater, 5
- Camera, 7
  - addOrUpdateDict, 7
  - Camera, 7
  - GetBeliefObject, 8
  - GetBeliefProbability, 8
  - GetPossibleTrafficSigns, 8
  - GetTrafficSignsVector, 8
  - UpdateTrafficSigns, 8
- GetBeliefObject
  - Camera, 8
- GetBeliefProbability
  - Camera, 8
- GetCurrentAction
  - Navigation, 10
- GetCurrentSpeedProbabilities
  - Speed, 15
- GetIdentifier
  - AI::BeliefUpdater, 5
- GetNavigationVector
  - Navigation, 10
- GetObsProb
  - AI::BeliefUpdater, 6
- GetPlannedAction
  - Navigation, 10
- GetPlannedNavigationVector
  - Navigation, 10
- GetPossibleNavigation
  - Navigation, 10
- GetPossibleTrafficSigns
  - Camera, 8
- GetSpeed
  - Speed, 15
- GetSpeedStates
  - Speed, 16
- GetTrafficSignsVector
  - Camera, 8
- GetTransProbSum
  - AI::BeliefUpdater, 6
- InitialiseSpeedProbabilities
  - Speed, 16
  - Navigation, 9
  - AddOrUpdateDict, 9
  - GetCurrentAction, 10
  - GetNavigationVector, 10
  - GetPlannedAction, 10
  - GetPlannedNavigationVector, 10
  - GetPossibleNavigation, 10
  - Navigation, 9
  - PlannedToCurrentNavVector, 10
  - SetPlannedAction, 10
  - SetPlannedNavigation, 10
  - SetPlannedNavigationVector, 11
- Observation
  - AI::Observation, 12
- PlannedToCurrentNavVector
  - Navigation, 10
- RewardStateAction
  - AI::RewardStateAction, 14
- SetPlannedAction
  - Navigation, 10
- SetPlannedNavigation
  - Navigation, 10
- SetPlannedNavigationVector
  - Navigation, 11
- SetSpeed
  - Speed, 16
- SetUpdatedSpeedVector
  - Speed, 16
- Speed, 15
  - GetCurrentSpeedProbabilities, 15
  - GetSpeed, 15
  - GetSpeedStates, 16
  - InitialiseSpeedProbabilities, 16
  - SetSpeed, 16
  - SetUpdatedSpeedVector, 16
  - Speed, 15
  - UpdatedToCurrentSpeedVector, 16
- Transition
  - AI::Transition, 17
- UpdateClassifier
  - AI::UpdateClassifier, 18
- UpdatedToCurrentSpeedVector
  - Speed, 16
- UpdateTrafficSigns
  - Camera, 8





