POMDP Learning in Partially Observable 3D Virtual Worlds

Jonah Hooper
University of Cape Town
Cape Town, Western Cape, South Africa
jonah.graham.hooper@gmail.com

ABSTRACT

In this paper we describe PCog - a system for POMDP based model learning and test the validity of that method in a 3D virtual world. The method used is based on Utile Suffix Memory which is used to generate an Markov Decision Process (MDP) which is then converted into a Partially Observable MDP using a series of transformations. PCog exhibited positive performance characteristics in comparison to a BDI based agent and manually defined POMDP. PCog is a novel application of model based POMDP learning to a real time system.

KEYWORDS

POMDP, 3D Virtual Worlds, BDI, Partial Observability

1 INTRODUCTION

This paper addresses the problem of stochastic planning and acting in a 3D real time game environment using partially observable markov decision processes. The challenging aspect of these sorts of environments is that their large state spaces are not easily modelled as Partially Observable MDPs. Concretely this paper extends a technique of off-line model based POMDP learning to derive POMDPs on the fly in real time 3D Unity[6] based testing environment. The environment is a CAIR[1] lab project called QCog. The entire system used to perform the model learning and planning will from here on be known as PCog.

Other reinforcement learning techniques that are used to solve this problem include model-free value and policy iteration. In addition there is also model based learning that attempts to learn parameters for markov models with defined states. This paper focusses on model based learning where an agent can only observe some part of it's environment and where it's state space is large, complex and unknown to the agents designers. We do however assume that this environment can be modelled using Markov Decision Process. In addition the notion of partial observability and degree of belief are also discussed in this paper.

There are were two primary aims of this project - to extend the QCog architecture to support planning with partially observable Markov decision processes and to assess whether those extensions provide reasonable performance improvements when compared to other techniques. In addition to agent performance we consider the tradeoffs of the technique from the perspective of an agent designer.

In order to solve the problem above this paper used a technique of MDP learning called Utile Suffix Memory (USM)[3.2] and adapted it to create a POMDPs[2.5]. An agent learns the states of it's world through observations it receives as it interacts with its environment. In addition the generated POMDP is refined as the results of taking actions using the generated POMDP are fed back into the USM module and used to create better POMDPs.

This method has been shown to perform comparatively to a belief desire intention agent and a hand defined defined POMDP. PCog also performs far better than the default Q-Learning based agent in Q-Cog. Our method also allows for an agent designer to have less knowledge of their environment than the BDI and static POMDP approaches.

This paper will start with the background section covering the basics of reinforcement learning, the QCog framework and Utile Suffix Memory. It will show the overall design of PCog in the design section and elaborate on more specific implementation details in System Implementation. Results, discussion of qualitative observations are found in the Results and Discussion sections respectively.

2 BACKGROUND

2.1 QCog

QCog is an architecture designed for adaptive self-learning agents in 3D environments that are both complex and unpredictable. It is owned by Centre for Artificial Intelligence Research (CAIR)[1]. It is designed to be an experimental platform for agent development and evaluation using the Unity3D Game Engine[6]. The unity game is a test harness used to assess the abilities of agents. It will be referred to as **test bed** from now on. Currently QCog contains a reinforcement learning mechanism that uses a dynamic policy selection mechanism that enables a cognitive agent to adapt to unknown situations in its environment[14]. Entities placed in the simulated world are designed to be generic and extensible for ease of use within the architecture. QCog features adjustable simulation settings which can be used to set the number of iterations you wish to perform, a simulation speed control mechanism, a data recorder which records important data metrics during the simulation, and finally a playback engine that allows simulations to be recorded and played back for further study whenever the user desires[14]. In the testbed the agents vision is restricted to a 180 degree hemisphere and it it often has it's vision obscured by obstacles. These restrictions add an aspect of partial availability to the agent.

2.2 Belief Desire Intention Architecture

Belief Desire Intention (BDI) is an architecture for designing and constructing intelligent agents[7]. It is based on the Belief Desire Intention model of human cognition[7]. Beliefs are the current model of the agents state. Desires are long term goals that the agent may wish to achieve. Intention is the current desires that are being prioritised for achievement.

2.3 Markov Decision Processes

Markov decision processes are models of stochastic environments in which reinforcement learning takes place. MDPsconsist of a discrete set of states S, a set of actions A and a set of conditional probabilities between actions and states known as T - the transition function. More formally MDPscan be defined as a **4-tuple** of the form[2]

$$MDP = (S, A, T, R, s_0) \tag{1}$$

where

- S - the set of states
- A - the set of actions
- T - T is a function where $T(s, a, s')$ is the probability of transitioning to state s' given action a and state s.
- R - R is a function where $R(s', a)$ is the reward for taking action a and arriving in state s'
- s_0 - the starting state

2.4 Reinforcement Learning

Reinforcement learning is a subset of the larger field of machine learning which typically deals with training artificial intelligence agents to act more effectively in their environments. More concretely reinforcement learning can be expressed as the problem of controlling an agent so that it acts optimally in a Markov Decision Processes (MDP)[20, pp 4]

MDPs are extensions of Markov Models[16] which are environments in which there is a transition function T which jumps from state to state according to some conditional probability distribution $T(s'|s)$ where $s', s \in S$. The difference between Markov Models and Markov Decision Processes is that an agent in a Markov Decision Process can interact with its environment using actions.

2.4.1 Model Based and Model Free Learning. There are two primary types of reinforcement learning - model based and model free learning. Model based learning involves learning a model (typically an MDP) of the environment perceptions of the agents state, when taking actions actions and receiving rewards[18]. An agent will typically learn a transition $T(s, a, s')$ and reward $R(s, a)$ which are the transition and reward functions respectively. Under this scheme the agent must assume that the state of the model is known ahead of time and is fully observable. Model free learning is the process of learning a value function - that is some function $Q(s, a)$ that represents the action in state s that will result in the highest reward over time. Model learning assumes only a state space - it does not require an explicit model of the transition function or reward function.

2.4.2 Value Iteration. Value iteration is model free learning that involves successively approximating a better value associated with each state and possible action [**?** , p. 109] In other words value iteration involves approximating a function $Q : S \times A \longrightarrow \mathbb{R}$ which is the best expected long term reward of taking action a in state s.

2.4.3 Policy Iteration. Policy iteration is a model free learning technique. A policy - $\pi(s)$ - is a sequence of actions that could be taken from state s. Policy iteration will iterate through possible policies until it finds $\pi^*(s)$ which is the optimal sequence of actions to take from state s.

2.4.4 Q-Learning. Q-learning is an algorithm for value iteration that was used for the primary agent demonstrated in QCog[14]. It is the reason for the Q in **Q**Cog. Q-learning is summarised by the following formula[10]:

$$\begin{aligned} Q(s_t, a_t) \longleftarrow & Q(s_t, a_t) + \\ & \alpha[r_t + \gamma max_{a \in A} Q(s_{t+1}, a) \\ & - Q(s_t, a_t)] \end{aligned}$$

t represents the time epoch. s_{t+1} is a successor state to s_t. $Q : S \times A- > \mathbb{R}$ is the expected value of taking action $a \in A$ in state $s \in S$. An agent using Q-learning will maintain a Q-Table [10] which is a data structure that stores the values of the Q-table. The table is updated at each time epoch using the Q function. QCog - the java code base MessageClient - PCog client in the code base TestBed - the Unity test application PCogMessageHandler - python module that interprets messages from the PCogClient ModelLearner - a state machine that dictates exploration and exploitation strategy of PCog. It also maintains the Utile Suffix Memory model PerceptionProcessor - this module takes raw perceptions coming from the MessageClient and discritises them into a form that is compatible with the reinforcement learning module. Also contains methods for scoring perceptions - IE a reward function. It also contains an exploration strategy.

2.5 Partially Observable Markov Decision Process (POMDPs)

A POMDP is a model of a stochastic process where the current state of the agent is unclear. A "belief state" is maintained about the POMDPs environment which is a probability distribution over a set of states. The belief state is often denoted as $b(s)$ where s is a state and $\sum_{s \in S} b(s) = 1$ where S is the set of states. The agent is able to infer beliefs on it's environment by making use of the idea of observations. Observations are emitted when an agent arrives in a new state having taken some action. Observations allow the agent to update it's belief state based on new information.

Formally, a POMDPcan be defined[5, 12] by a **7-tuple** of values -

$$P = (S, A, R, Z, T, O, b_0) \tag{2}$$

- S - set of states
- A - set of actions
- Z - set of observations
- R - $R(s', s, a)$ is a function that maps every state action pair to a real number. This is the reward of taking action a in s and arriving in s'.
- T - $T(s', a, s)$ is the probability of transitioning to state $s' \in S$ given action $a \in A$ and incident state $s \in S$
- O - $O(z, a, s')$ is the probability of observing $z \in Z$ given action a and state s'.
- b_0 - a probability distribution over S that defines the starting belief state.

2.6 POMDPscompared to MDPs

Recall that MDPs2.3 are stochastic processes where an agent can make decisions that effect their environments. POMDPs are a generalisation of the idea of MDPs. They allow for the admission of partial information to the underlying model of a MDP. A POMDP model assumes that a system has an underlying MDP but the current state of the agent is only accessible through observations. This lack of information is represented using the belief state. This addition does add complexity when attempting to solve the model. The task of "solving" or "planning" is the process of finding a sequence of actions $a \in A$ which lead to the highest amount of reward over either finite or infinite number of future actions. Since the current state of a POMDPis not known, the state is represented as b, the belief state, which is a probability distribution. There are infinitely many possible belief states. This is in contrast to MDP which have finitely many possible understandings of where it is in it's environment - it's current state $s \in S$. This difference allows POMDPs to be more general than MDPs but presents additional computational and conceptual hurdles when attempting to find an optimal policy for an environment. Methods for finding optimal policies in MDPs are typically not usable for POMDPs because they are computationally expensive[4]. Exact algorithms rely on constructing an MDP from a POMDP by representing possible future belief states as a piecewise linear convex function(PLC). When planning in a POMDP there are many possible belief states that can be generated from decisions made in the POMDP. The number of belief states can increase exponentially[5] as we look further into the future. This processes is computationally expensive since it involves solving a linear programming problem[4] to find the best PLC function. POMDPsdo not distinguish between actions that reveal information about the state of the agent and actions that affect the environment by causing the agent to transition to another state. As such a POMDP implicitly encodes a "cost of information" in it's model - a concept that does not exist in MDPs. In other words - efficient POMDP planning will often result in an agent taking additional actions that reveal more information about it's environment so that it can be more certain about the cost of taking future actions.

2.7 Belief State Update

In order to plan under a POMDP the agent must update it's belief state for ever action and observation that it takes. This belief state update can be expressed in the following way:

$$b_n(s') = \frac{O(s', a, o) \sum_{s \in S} T(s, a, s') b_{n-1}(s)}{Pr(o|a, b)} \tag{3}$$

This represents the degree of belief that we are in $s \in S$ given that we took action $a \in A$ and observed $o \in Z$. Further we can define a function called *State Estimator* (**SE**). State Estimator finds the next belief state given that an agent in the POMDP model took a and observed o.

$$SE(b, a, o) = b_n \tag{4}$$

The reward function of a POMDP provides feedback on taking an action a in state s and arriving in state s'. But when planning in a POMDP and agent is not aware of it's current state. It only has access to a belief state b_n which is a probability distribution. Hence the expected reward for taking action a given belief s is

$$\sum_{s \in S} R(a, s) b_n(s) \tag{5}$$

When planning in a POMDP we are interested in finding an action that maximises reward over some given, possibly infinite horizon. The value of the best action is given by

$$\begin{aligned} V^*(b, h) = \max_{a \in A} & \sum_{s \in S} R(a, s) b(s) \\ & + \gamma \sum_{o \in Z} O(o, a, s') V^*(SE(b, a, o), h - 1) \end{aligned}$$

where $\gamma \in [0, 1]$ is known as the discount factor. The discount factor is used to weight the importance of future actions in the value function calculation. $\gamma = 1$ forces future actions to be weighted equally to current actions and $\gamma = 0$ states that only the immediate action is considered in the reward calculation. $h \in \mathbb{Z}^+$ is the horizon that dictates how far into the future we should plan. The function $Q^*(b, h)$ gives the action that maximizes future discounted reward. It is represented as

$$\begin{aligned} Q^*(b, h) = \text{argmax}_{a \in A} [& \sum_{s \in S} R(a, s) b(s) \\ & + \gamma V^*(SE(b, a, o), h - 1)] \end{aligned}$$

2.8 Monte Carlo Approximations

Since PCog has the potential to contain a large state space (discussed in the design section) being able to solve POMDPs with large state spaces is required. In practice Utile Suffix Memory can create POMDPs with large state spaces as well. Traditional **POMDP** solution algorithms like value iteration and policy iteration suffer from a curse of dimensionality. [Silver and Veness] proposes a ground-breaking POMDP approximation algorithm known as POMCP. Makes use of both particle filters and Monte-Carlo Tree Search (MTCTS) to approximate a solution to a given POMDP. The algorithm can compute policies which approximate the maximum expected utility to arbitrary error $0 < \epsilon$. POMCP constructs a history node h for a sequence of actions $a \in A$ and observations $o \in Z$. Each history will refer to a node in a search tree. The POMCP algorithm assumes some generative model (o, s', a)~$G(s, a)$. G is used to sample actions, states and observations. The algorithm selects the action to take using following equation:

$$\text{argmax}_{a \in A} V(bh) + c\sqrt{\frac{logN(h)}{hb}} \tag{6}$$

where

- $V(bh)$ is the value function of node h given belief state b
- $N(h)$ is the number of visits to node h
- c is a discount factor

POMCP assumes that the exact belief state can be found from a given history $h \in H$ and state $s \in S$. POMCP uses a particle filter to appoximate a value for $B(s, h)$ to avoid state estimation function (recall SE) which can be intractable for extremely large states.

2.9 Utile Suffix Memory

Utile Suffix Memory is a method for learning Markov Decision processes from a set of actions and observations. It has also been shown[17] that POMDPs can be derived from USM trees with some simplifying assumptions. **USM** assumes that an agent is operating in an environment where it can perceive observations ($o \in Z$) and can take actions ($a \in A$). Utile Suffix Memory organises its actions and observations in a suffix tree data structure from which it can derive an MDP.

For each action the agent takes it receives an observation and a reward. The *ith* action, observation and reward triple is given by (a_i, o_i, r_i). These tuples are organised into a series of *instances* which are a linked list where the triple at time t is linked to previous triples. Instance at time t is given by

$$T_t = < T_{t-1}, a_{t-1}, o_t, r_t > \tag{7}$$

Utile Suffix Memory is a suffix tree constructed from the suffixes of given instances. The suffix tree will contain actions in the odd rows and observations in the even rows. Older observations of the USM tree are stored at the top of the tree. Each node in the suffix tree contains a set of instances which have suffixes that led them to that node in the suffix tree. Nodes in the tree act as buckets for instances that reach those tree nodes. The deeper a node is in the tree the father back in time it occurred.

To ensure more efficient tree expansion (McCallum) calls for maintaining a *fringe* bellow the suffix tree. A fringe node contains references to sequences of instances that contain a prefix that is equal to a preceding suffix for some window. The leaves of these fringes are not initially considered to be part of the official leaves (states) of the tree. The leaf nodes of the fringe are made into states of the USM if there is a statistically significant difference between the expected values (see equation (10)) of their instances and that of their parent official leaf. An example of a USM tree with suffixes and fringe nodes can be seen in figure (1).

In order to insert a value in the tree the USM insertion algorithm will trace n preceding instances of some instance T_t. There are three possible outcomes that can occur when inserting an instance into the tree:

- A fringe element s is reached - s is promoted to a full leaf node and state. All of s ancestors are no longer fringe nodes
- The a leaf of the tree is reached - in this case a new state is added to the tree and the tree is extended to include T_t
- An internal non-fringe node s is reached - T_t is added to the bucket of instances in node s

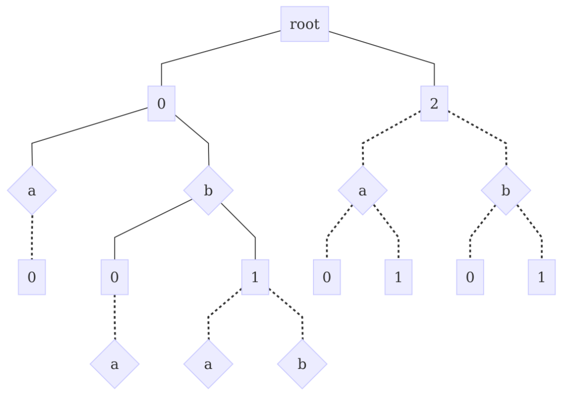

Figure 1: Example of a Utile Suffix Memory tree with fringe nodes connected by dotted lines

To construct a USM tree the agent initially will set out to explore in it's environment. The agent will record what it sees and will add instances to the suffix tree. The leaves of the suffix tree become the states of the agent.

The leaves of the tree can be referred to as s . Each leaf can also be used to refer to the suffix of instances that was used to travel from the root of the tree to the leaf. We denote the leaf node of instance T_i as $L(T_i)$. We denote the set of all instances associated with leaf node s as $\tau(s)$. All instances associated with s that contain action a are denoted by $\tau(s, a)$.

The immediate reward of being in state s and taking action action a can be given by

$$R(s, a) = \frac{\sum_{T_i \in \tau(s,a)} r_i}{|\tau(s, a)|} \tag{8}$$

In other words the reward of taking a in s is the average reward of taking a while attempting to reach state s as the agent makes observations.

Similarly the transition function is given by

$$T(s', a, s) = \frac{|\forall T_i \in \tau(s, a)\; L(T_{i+1}) = s'|}{|\tau(s, a)|} \tag{9}$$

Intuitively this represents how many possible points there were in the suffix tree wherein action a was taken and led to the next instances landing up in tree s' instead of s .

The value function of a USM tree can be found in the following manner

$$Q(s, a) \longleftarrow R(s, a) + \gamma T(s', a, s)U(s') \tag{10}$$

where $U(s, a)$ is the utility function of the Utile Suffix Memory. U is given by

$$U(s') = \max_{a \in A} Q(s', a) \tag{11}$$

The performance of calculating Q can be sped up using dynamic programming. Q is similar to simple Q-learning of a MDP.

To expand fringe nodes underneath leaf nodes [McCallum] recommends the following procedure:

(1) For each official leaf node of the USM tree - in other words all states $s \in S$
(2) For each child node c of leaf s
(3) For each instance - T_i - in the child c that is a fringe node, calculate it's utility using $U_f(T_i)$
(4) Calculate the utility of each instance in the official leaf node s using $U_f(T_i)$
(5) Compare the two distributions using a two sampled KS test
(6) If the distributions are different (we reject the null hypothesis of KS test) promote the fringe node to an official leaf node
(7) Repeated the process until no new fringe nodes can be promoted to leaves or there are no fringes left

$U_f(T_i)$ is taken to be $U(T_i) = r_i + U(L(T_i))$ where $L(T_i)$ is the leaf node associated with instance T_i (if this node is in the fringe then this would be the last leaf node one would encounter while traversing to the given fringe node).

This procedure ensures that a fringe node is only "promoted" to a leaf if it adds enough new information that justifies adding it as a state.

2.10 Planning in USM

In order to plan in a USM tree an agent must choose the action that has the highest expected value. The next best action (a^*)is given by

$$a^* = \text{argmax}_{a \in A} Q(L(T_i), a) \tag{12}$$

where T_i is some instance and $L(T_i)$ is the leaf node associated with that instance.

3 RELATED WORK

3.1 POMDP Based Model Learning with Braum Welch

[Koenig and Simmons] suggest using a hill climber algorithm that learns a sensor model $Pr(o|s, a)$ that would most likely generate a sequence of observations and actions ($\{o_t, a_t, o_{t-1}, a_{t-1}, ...\}$) that were observed by the agent. The approach assumes that there is a known state space in the model $s \in S$ that the agent already knows.

3.2 Model Based POMDP Learning using Utile Suffix Memory

(Shani) describes a method of converting USM into POMDPs. The state space (S) of the POMDP is given by the set of all leaves of the USM suffix tree. The observation (Z) and action (A) spaces are defined as part of the environment. Under this scheme the transition function of the model is given by 9. The reward function ignores the effect of the arrival state but still constitutes the reward function for a valid POMDP. It is given by 8.

Defining an observation function for USM requires the notion of sensor accuracy. Sensor accuracy is defined as $Pr(o = o')$ which can be interpreted as the probability that observation o could actually be an observation o'. This is a slightly weaker requirement than that of the observation function of a POMDP which requires a mapping between states, actions and observations but is still sufficient to construct a valid POMDP. (Shani) suggests deriving O from the USM in the following manner

$$O(a, s, o) = Pr(o = o(s)) \tag{13}$$

where $o(s)$ is the last observation associated with the tree node for state s. o has the signature $o : S \to Z$ where Z is the observation space. So $Pr(o = o(s))$ is the probability that observation o would be viewed immediately before reaching state s.

3.3 Choice of Learning Algorithms

Utile Suffix Memory based approach was chosen instead of the Baum Welch because it does not require an agent designer to explicitly model the state space of the environment. This is in contrast with Baum Welch which does require a state space to be specified. With the Utile Suffix Memory approach an agent designer would only have to design a reward function for the agent and not have to design a state space.

4 DESIGN OF PCOG

4.1 Overview

The primary constraints that drove the design and architecture of the PCog agent was that the most compelling POMDP library, AI-toolbox[21], was only usable from C, C++ and python. In addition there was already logic implemented to process messages from the test bed in the Java based QCog agent. To simplify the implementation and to reduce possible avenues for introducing new bugs and errors in communication between a Python based agent and the test bed - the QCog framework was extended to act as a intermediary between PCog and the test bed. The QCog module (Humanoid) maintains agents view of the environment. PCogClient extracts that information and passes it on to the PCog python code base.

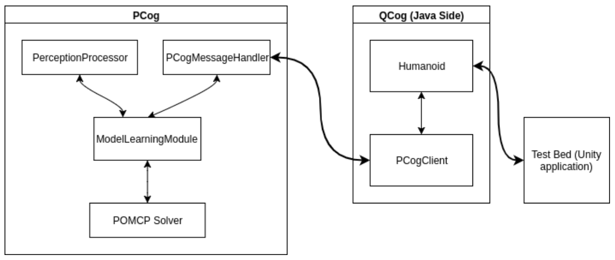

Figure 2: A diagram of the flow of information within PCog

As a consequence of the PCog related extension, we extended QCog to allow it to communicate with external processes which can advise it how to plan and act. These extensions allow an arbitrary agent on an arbitrary computer to communicate with the testbed or form a component of the QCog architecture. This design choice was made to exploit the maintenance of the testbeds state which was handled by the Humanoid class in QCog.

The figure 2 provides an overview of the PCog architecture. Some of these components are covered in more detail later on.

- **QCog** - the java code base
- **Humanoid** - a singleton class in QCog that maintains the agents idea of the testbed.
- **PCogClient** - sends and receives messages over a TCP-Socket between java code base and python based PCog agent
- **TestBed** - the Unity test harness application
- **PCogMessageHandler** - python module that acts as a TCPServer that listens for messages for new state changes in the PCog-Client.
- **PerceptionProcessor** - this module is defined by the agent designer. It contains functionality to convert the raw information from the test bed into a discitised representation. This module also defines the observation space Z. It accepts state received through PCogMessageHandler and returns instances which can be inserted into the USM Module in the model learning agent.
- **ModelLearner** - a state machine that handles POMDP model generation and agent exploration and exploitation. Also maintains the Util Suffix Memory.
- **POMCP Solver** - a thin layer on top of POMCP algorithm implemented in AI-toolbox. Accepts descriptions of POMDPs from the test bed and returns the action that best maximises the description.

4.1.1 *Message Client - PCogClient.*
PCog client is an extension to the QCog architecture that allows the offloading of decision

making to an external agent accessible through a TCP socket. PCogClient sends immutable snapshots of the current state of the test bed according to the QCog agents perception (which is stored in the Humanoid class). The immutable subset of the code would be converted into JSON and sent over to PCog. The immutability of these created snapshost also allowed the PCogClient to analyze state changes in the environment. The messaging client also abstracts the notion of time in the testbed away from PCog. This is discussed in more detail in 6.1. PCogClient is also paired with a TCP server - PCogMessageHandler - on the python side of PCog which processes messages received from PCogClient.

4.1.2 *Perception Module.*

The perception layer mirrors the perception layer created by (Michael[14]). The perception layer accepts the testbed state from the PCogClient and converts it into the observation space. In other words the testbed takes some world state W and converts it into $o \in Z$. W can be seen as the current state of the testbed which is extremely large and possibly continuous. Perception is considered an interface in PCog.The perception engine contains a reward function for the observation. This is based on the world state W rather than o. This is used to create r_i for for $(a_{i-1}, o_i) \in A \times Z$. This is used to create instance T_i. This allows an agent designer to prioritise specific behaviors and results that PCog can observe. In other words a Perception Module can be defined in the following way:

- $R : O \times O \longrightarrow R$ - a reward function based on the current and the previous observations
- $\Pi_e : W \longrightarrow A$ - an exploration policy that maps the current world state to an action
- $Perceive : W \longrightarrow O$ - an optional function that takes some complex state - W - from an external source and converts it into an observation O

A designer of the agent will typically implement all 3 of those functions.

4.1.3 *Utile Suffix Memory Module.*

The utile suffix memory module contains an implementation of the Utile Suffix Memory. The utile suffix memory module maintains the following primitive states

- Z - a set of allowed observations
- A - a set of allowed actions
- I - root node of the instance linked list
- R - the root of suffix tree
- S - set of suffix tree leaf nodes
- w - number of instances to consider part of history when inserting into the tree. In other words the instance suffix length. The tree height would thus be $2w$ without using fringe insertions.

This module handles the insertion of new perceptions into the tree and the creation of the POMDP from the perceptions.

4.1.4 *Model Learning State Machine.*

The model learning state machine is an abstraction that handles the exploration, generation of models and maintenance of the USM. The model contains the following hyper parameters

- ϵ - an exploration vs exploitation constant
- U - a utile suffix memory
- m - how many iterations between each update of the learned POMDP model
- P - POMDP model derived from U. This is initially non-existent
- O - a perception module

This module was created because it provides an abstraction barrier between the storage of data - USM Module - and the training strategy for the agent. It also ensured that PCogMessageHandler could be free from handling logic associated with the agents decision making.

4.2 Software Libraries

The AI-toolbox library[21] was used to solve the constructed POMDPs. Using a library was a design decision that was taken to increase development time and reduce the likelihood of mistakes being made in POMDP solving. This particular library was chosen because it had a wide range of POMDP solving algorithms implemented like Incremental Pruning[3], POMCP[19] and Witness Algorithm. It also presented a simpler programming model than other libraries like pemani4911 and [22] since it does not require the creation of a complex class hierarchy to construct a model which stands in contrast to the other two libraries mentioned. Only vectors of floats are required to specify a POMDP model in Svalorzen.

4.3 Discretising Environment

The QCog test bed is a real time environment[14]. The test bed sends a stream of updates on it's environment state which were passed to the QCog java API. Extensions were made to the QCog java API (Cite Yusri and Wills work) that allowed the agent to detect whether an action that it made complete. The actions effected by this were the *FLEE*, *ATTACK* and *EAT* (6.1) actions. PCogClient added support for running the explore action until an entity was seen in the test bed. This allowed the agent to view actions in the world as a series of discrete synchronous state changing commands.

5 SYSTEM IMPLEMENTATION

PCog is a POMDP model learning framework that interlops with the QCog environment. The basic idea of model learning is that the agent will spend some time "exploring" it's environment. In the case of PCog "exploration" is some policy $\Pi_e(o) \in A$ where $o \in W$ and W is the world space of the testbed and where A is the set of allowed actions available to an agent. The agent will spend some time taking actions in it's environment. During this period the agent constructs a Utile Suffix Memory U.

Once the agent has perceived enough of it's environment it will derive a POMDP - P - from U using a function G (G for generate). The concept of "enough" of the environment is defined as a set number of iterations. This idea could be defined in future work. The agent will then follow a policy called $\Pi_{U,P}(o, \epsilon) \in A$ which is a composite policy derived from two policies: Π_e and Π_P. $\Pi_e(o) \in A$ is an arbitrary policy that can be specified by an agent designer. In the case of PCog it is a set of rules that the agent follows. $\Pi_P(b_h)$ is the optimal policy derived from planning in POMDP P. We pick Π with a probability of $0 \leq \epsilon \leq 1$ We construct b_h from some history of past perceptions $h \in H$ where H is a list of previously observed perceptions.

In other words $\Pi_{U,P}(o) \in A$ can be defined in the following way

$$\Pi_{U,P}(o, r) = \begin{cases} \Pi_e(o) & \text{if } \epsilon < r \\ \Pi_P(B(o)) \end{cases} \tag{14}$$

where $r \in [0, 1]$ is a random number sampled from the uniform distribution U and $B : H \longrightarrow \Delta(s)$ where $\Delta(s)$ is the set of all probability distributions over S.

5.1 Exploration and Model Refinement

Once a new POMDP was generated from Utile Suffix Memory the agent will plan off that POMDP (P). In addition to planning off P the agent will also take an action from a specified exploration (recall $\Pi_{U,P}$ (5)) policy with probability ϵ. This step allows the agent to discover new sequences of actions. Currently the *epsilon* value does not change. Varying the value of $\epsilon \in [0, 1]$ according to some exploration strategy could be seen as an instance of the multi-armed bandit problem and could be explored in future work.

By sampling actions from the P and adding them to the utile suffix memory instance we allow regressive patterns developed by P to be corrected when a new POMDP is generated. In this way we are iteratively improving the performance of the P that gets generated. Regeneration of P is controlled by a parameter known as m. Every m perceptions a new POMDP is generated and deployed. From experimentation it was found that too many regenerations caused performance regressions(8.4)

5.2 Planning with the POMDP

Once P_i has been constructed from the USM tree it's initial belief state still needs to be found so that planning can be performed on the POMDP. The initial belief state of the POMDP was found by traversing the most recent sequence of instances from the root tree node to find the current state of the agent in the suffix tree. If no leaf state is found then the a uniform distribution over the set of possible successor states to the last internal tree node is chosen as the initial belief state. Once the belief state is found the POMDP is solved using the POMCP algorithm2.8 implemented in AI-Toolbox[21]. The derived POMDP is used as an approximation of the world. The agent then performs planning on this approximation to find the next best action.

5.3 Exploration Strategies

Before a POMDP can be derived from the utile suffix memory the agent must explore it's environment and discover new states. Explore in this context does not refer to the *EXPLORE* action discussed in 6.1 but rather to a policy (recall Π_e) that the agent uses to discover new information about it's world. Two explorations policies were considered by the agent - the first was a "smart exploration" strategy. Smart exploration policy was a hand written strategy that provided some intelligence to the agents behavior. For example the strategy would guide the agent to eat berries when they are seen or attack a wolf when one is seen. The second exploration policy - random exploration - picks actions at random regardless of the state of the environment. Use of the smart explore policy allows the agent designer to encode their intuitions of the environment into the final POMDP that is produced by PCog.

5.4 Observation Space

The allowed observations for the state space is given by the following 7-tuple:

- $predator\_visible \in \{0, 1\}$ - 1 instructs the agent that a predator is visible
- $food\_visible \in \{0, 1\}$ - 1 instructs the agent that food is visible
- $health\_level \in \{0, 1, 2\}$ - the level of the agents health with 0 being the lowest level
- $movement \in \{0, 1, 2\}$ - amount the agent moved since the last action with 0 being stationary, 1 being medium amount of movement and 2 being a lot of movement
- $died \in 0, 1$ - did the agent die in this observation - with 1 being that the agent did die

The mechanism used to derive the observation function in PCog differs from the one suggested by [Shani] since the test bed framework has no notion of sensor accuracy (recall $Pr(o = o')$). In order to handle this the observation function of a POMDP derived from Utile Suffix Memory is defined as

$$O(s, a, o) = \begin{cases} \frac{|\{\tau(s,a) | Obs(T_i) = o\}|}{|\tau(s,a)|} & \text{if } |\tau(s, a) > 1 \\ \frac{1}{|Z|} \end{cases} \tag{15}$$

where S is the set of all leaf nodes that are also states. This way an observation function can be taken to mean "If we reached state s how often would we view o having also taken action a on the way". In the case that no actions have been observed to reach state a then since we have no information we assume that every observation is equally likely to be viewed in s and action a.

6 EXPERIMENT DESIGN

The QCog test bed was used to test the performance of the agents. The test bed originally designed designed as a real time test bed to assess the performance of Q-Learning based agents in a real time test environment. Some adaptations and simplifications were made to make it easier to model the environment as a POMDP.

6.1 Scenario Description

The test scenario centers around an agent which explores it's environment and attempts to survive. The agent maintains health level $h \in \{0, ..., 10\}$. The agent dies if $h <= 0$. The test scenario contained 6 predators which would attack and damage the agents health when they were close to the agent. Each of the 6 predators attacks caused different damages. There were also 4 berry bushes which the agent could eat to increase it's health. The agent was allowed to take 4 kinds of actions in the test bed -

- *EXPLORE* - A random walk around the map. Does not attack or eat any entities. Must not be confused with Π_e which is a policy.
- *ATTACK* - attacks the closest predator
- *EAT* - eats the closest berry
- *FLEE* - flees one of two predefined waypoint

The test bed communicates with client agents in real time. This is in contrast with a model where the test bed waits for inputs before updating it's own internal state of the world. When a client agent sends a message to the testbed, the agent will attempt to complete that action regardless of whether it was busy completing a different action. The real time nature of this environment made it more difficult to model since the time taken to process a decision would have an effect on the performance of the agent. In order to deal with this caveat some effort was taken to ensure that actions would run until "completion". PCogClient allows *ATTACK*, *EAT* and *FLEE* to run until an "action complete" flag is sent from the test bed or until some timeout is reached. *EXPLORE* runs until the agent sees some entity (Berries or Predators).

Partial information in the test bed is encoded by the agent only having a forward facing hemisphere for it's field of vision. Other potential avenues of partial observability that were not accounted for was the differing strength of the wolves.

Runs of the testbed are divided into simulations and iterations. Iterations are a single run of the scenario until the agent is killed or kills all the predators. Agents are allowed to retain learned information between iterations. Simulations are groups of many iterations. Agents cannot retain learned information between simulations.

6.2 Experiment Details

We tested PCogs performance against 4 different agents

- **RCog** - An agent that picks actions at random. This agent provides a lower bound for performance
- **BCog** - A Belief Desire Intention[9] based Agent with probabilistic notions.
- **QCog** - Q-Learning based agent described by (Michael). This is the default agent in the test bed.
- **spcog** - A manually defined POMDP

The agents were scored according to a scoring system developed by (Grant). The following metrics were used:

- G - 1 if the agent killed all predators, otherwise 0
- T - length of time the agent survived for
- A - number of attacks that did damage / number of attacks
- P - number of predators killed / total predators
- F - number of times agent successfully fled / total number of time fled

These metrics are combined in the following way to get the score metric

$$S = G + T\frac{A + P + F}{3} \tag{16}$$

In addition to score we used number of kills and survival time to measure agent performance. A kill is when an agent attacks a predator causing its health to drop bellow zero. Survival time is the time the agent took to complete an iteration. The iteration ends when an agent dies or kills all the predators.

We thought that these metrics best capture the agents performance with score being a more find grained metric.

Each agent was run for 3 simulations. Each simulation contains 20 iterations. Making a total of 60 iterations per agent. The results were recorded by the build in metrics system.

7 RESULTS

In this section we discuss some quantitative metrics of the agents performance. The metrics chosen for discussion were score (recall S), number of kills and survival time. Bellow is a table of the mean performance of the agents across all recorded iterations for the selected metrics:

Table 1: Mean agent perfomances

Agent	PCog	SPCog	BCog	QCog	RCog
Kills	4.75	4.45	5.01	2.5	0.1333
Score	1.22	1.034	1.545	0.77	0.43
Survival Time (s)	257.4	308.11	283.61	206.15	87.86

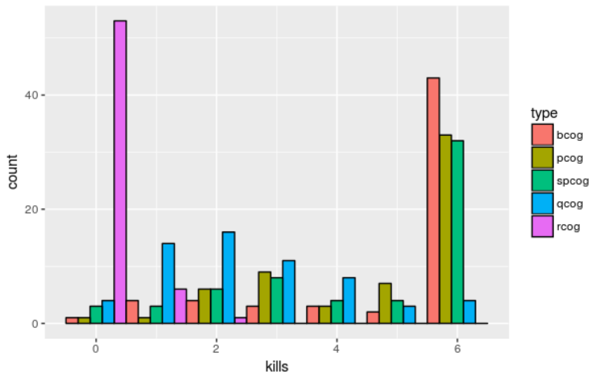

Figure 3: Histogram of number of kills for each agent type

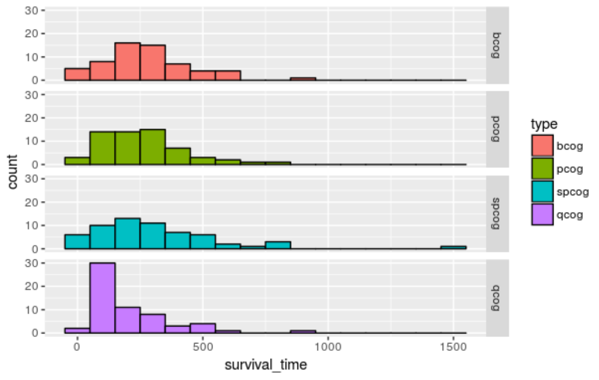

Figure 4: Histograms for the survival times for 4 of the 5 agents

7.1 Statistical Tests

Table 2: KS Test of score distribution - PCog vs other agents

Agent	SPCog	BCog	QCog	RCog
p	0.3777	7.57×10^{-6}	1.64×10^{-6}	9.992×10^{-16}
D	0.167	0.45	0.4883	0.7333

Table (2) shows the D-statistic and P-statistic of the score metric of all agents tested against PCog. We ran two factor non-parametric KS-test on each pair of metrics for all the agents. Running KS tests on PCogs kill rate revealed that the while BCog appears to have a higher kill rate but the difference in kill ratios between

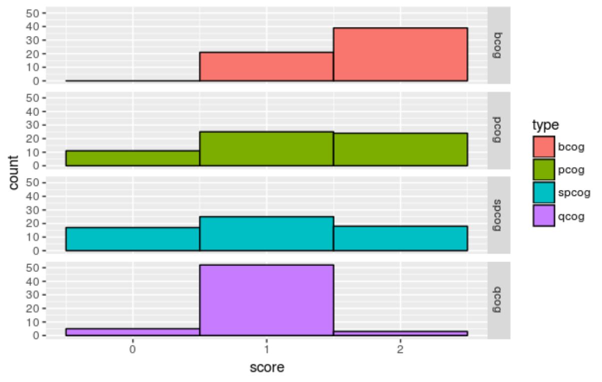

Figure 5: Stacked histograms of the scores of the tested agents

BCog and PCog is not statistically significant. In addition, the kill rates of SPCog and PCog are not statistically different which seems to explain why their scores appear not to be different(2). It was found that the SPCog, PCog and BCog all outperformed QCog in a statistically significant fashion. This confirms the visual intuition presented in figure (3).

The score metric of the agents revealed no statistically significant performance differences between PCog and SPCog. BCog out performed all other agents in score in a statistically significant fashion according to KS tests. A somewhat unintuitive result is that the scores of SCog and QCog are not from the same distribution according to the KSTest as one would expect from figure (3).

Overall BCog is the highest performing agent in score. This result is statistically significant with $p = 0.05$. PCog is second with the second highest score and kill rate.

All the agents have very similar survival times which appear to come from the same distributions - confirming visual intuition in figure (4).

The full analysis of the results was conducted in the R programming language and can be found here.

8 DISCUSSION

Ultimately the results of experiment are promising as PCog is concerned. The agent performs surprisingly well against agents that have more information about their environments. Model based POMDP learning has not been used to perform planning in a real time system of this nature. The results indicate that POMDP model learning is a viable approach to developing adaptive systems. That being said some strange and somewhat regressive behaviours were noticed which are also discussed in this section. The performance of PCog against QCog is not discussed in this section because QCogs performance was extremely poor compared to PCog (see table (1)). We suspect that this is due to a lack of tuning of the agent to a new test bed.

8.1 PCog vs Hand Crafted

PCog appears to perform slightly better than SPCog - a hand crafted POMDP. While the results are not statistically significant this is an encouraging sign for the algorithm. SPCog requires knowledge of the underlying environment in which the agent exists. It requires significant simplifying assumptions to make modeling the environment easier. For example - SPCog has only 8 states. It does not represent different health levels of the agent. SPCog can be said to be more brittle to change since it cannot be generalised to new environments. PCog requires only two things - reward and observations - to be specified. This allows it to construct its own internal representation of the environment. PCog requires less encoded knowledge about its environment in comparison to BCog and SPCog. BCog encodes a set of beliefs and goals in its design. An agent designer must have an idea of a good strategy to construct a BDI. In summary both SPCog and BCog both require designer of the agent encode a strategy and state space. PCog does not make that requirement. The POMDPs generated by PCog can also be reused in new scenarios. The best PCog POMDP that is developed could theoretically be used repeatedly for the scenario. PCog could also be viewed to generate a POMDP as an end goal.

8.2 Hyperparameters

The PCog has a significant number of hyperparameters which could have an effect on its performance. These include the type of exploration policy being used. Random exploration policy had a noticeable negative effect on performance when compared to using smart exploration policy. In addition - the fact that the test bed updates clients in real time means that changing event processing or running the agent on a different computer could result in different performances for the agent. Having many hyper parameters results in additional complications for the agent designer as hyperparameters introduce a large vector of possible performance regressions and new complexity for a designer to learn. Hyperparameters can also make the system more brittle and over specific to certain problems which leads into the next section. That being said BCog encodes strategic information into the agent design. SPCog the agent designer has to approximate an MDP for the environment before engaging in further learning. If viewed in the context of those configurations the many hyperparameters of PCog look like less of an issue.

8.3 Influence of Smart Explore

At present it is unclear how much of the score PCog achieved is a result of Smart Explore. A PCog agent would on many instances get stuck in loop of the same actions in the test bed. Smart explore would fix this regressive behavior by taking an action that gets the agent out of a behavior loop. How significant this effect has on the agents results requires more explanation. That being said smart explore is an essential part of the reinforcement learning strategy since it allows PCog to learn new sequences of actions to perform.

8.4 Overfitting

As PCog was running the POMDPs that get generate seemed to get larger and larger. In addition the agent appeared to repeat previous sequences of actions. One could argue that with too much training PCog was starting to overfit itself to specific sequences of experience it had seen. This is somewhat counterproductive because it means that PCog is not generalising but is rather copying. Future work could include studying this effect more.

8.5 Performance Issues

While observing the agents behavior in the test bed the following common patterns were observed that could help explain why the agent performs worse than BCog.

The agent would often get stuck in loops of states especially during repeated actions. To mitigate this behavior random actions from smart exploration were seeded with probability of $\epsilon \in [0, 1]$. These random actions would usually break the agent out of it's exploration loop.

The agent would often rapidly switch between different actions. This seems to be a consequence of the nature of USM algorithm being encoded in the POMDP. The algorithm learns a sequence of actions which have the best reward. In a sense the USM tends to learn policies rather than individual actions. As a consequence POMDP will alias a sequence of actions to the incorrect state - this issue could be mitigated with more state information about the test bed environment in future.

The agent tended to get more cautious - favoring exploration and fleeing - as the simulation went on. This is possibly due to the fact that the agent was penalised heavily when it died. Experimentation showed that more aggressive strategies were more appropriate in the scenario so this can account for a reduction in score.

Overall the state space of the test bed is highly complicated. (Shani) obtained most results from a far simpler test environment.

9 FUTURE WORK

POMDP based model learning is a relatively obscure field and has not been applied to training agents in a real time continuous domain such as the QCog testbed. Future work could include developing more intelligent exploration - perhaps using ϵ-greedy algorithm to find a balance between model exploration and model exploitation (IE planning off generated POMDP). In the current implementation of the POMDP based model learning framework a static reward function is defined for an entire epoch. Future work could include making the reward function dynamic and adaptive allowing for PCOG to be more adaptive to new environments. While high level goals should remain unchanged in complex test bed scenarios the best strategy will change quite steadily as the epoch continues. For example an agent should be more aggressive when there is only one predator remaining rather than many. The POMDPs generated by the model learning algorithm later on in the running of the epoch tended to be quite large. (Shani) suggests a method for merging USM tree branches to reduce the state space size but did not include it in their final implementation since the merging algorithm was rarely used. Future work could include implementing an algorithm that effectively merges tree branches that are similar to each other and do not provide a significant classification advantage. Or an improved POMDP generation algorithm that takes the similarity of states into account when developing transition functions. The reward function described in (8) does not make use of a discount factor. Neither is a discount factor used in the observation function mentioned in (15). Future work could investigate the use of discount factors on the performance of the agent. In summary, future could address some of the performance issues in the discussion section and create better strategies to balance the model based learning agents exploration and exploitation parameters.

10 CONCLUSION

In this project we developed PCog - a POMDP based model learning framework. We evaluated the performance of PCog in a 3D virtual world against a series of other agent architectures. The PCog POMDP model learning provides a POMDP model learning framework that allows an agent to approximate POMDPs for it's environment. PCog is - to our knowledge - the first application of POMDP based model learning to a real time 3D virtual world. The agent is able to outperform an MDP agent and perform comparatively to a BDI based agent when run in the testbed. Further PCog was able to outperform a hand written POMDP. This indicates that PCog - and model based online learning - is a viable approach for constructing POMDPs for a complex real time test environment. Further - it also shows that POMDP model learning using Utile Suffix Memory is a viable strategy for MDP learning in environments where the underlying MDP parameters are unknown, which is especially true in complex real time environments. That being said there are a number of performance issues and regressive behaviours that PCog exhibits. The results of the experiment show that POMDP based model learning is a valid approach for training agents that plan and act effectively in real time 3D virtual world with partial information.

11 ACKNOWLEDGEMENTS

Thank you to Dr Gavin Rens for guidance and feedback. Thanks to Professor Deshen Moodley for supervision. Thanks to William Grant and Yusri Dollie for their tireless development of the Q-Cog platform.

REFERENCES

[1] 2017. CAIR. (2017). http://cair.za.net/
[2] Oguzhan Alagoz, Heather Hsu, Andrew J Schaefer, and Mark S Roberts. 2010. Markov decision processes: a tool for sequential decision making under uncertainty. *Medical Decision Making* 30, 4 (2010), 474–483.
[3] Anthony Cassandra, Michael L Littman, and Nevin L Zhang. 1997. Incremental pruning: A simple, fast, exact method for partially observable Markov decision processes. In *Proceedings of the Thirteenth conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc., 54–61.
[4] Anthony Rocco Cassandra. 1998. Exact and approximate algorithms for partially observable Markov decision processes. (1998).
[5] Anthony R Cassandra. 1998. A survey of POMDP applications. In *Working notes of AAAI 1998 fall symposium on planning with partially observable Markov decision processes*, Vol. 1724.
[6] Unity Game Engine. [n. d.]. Unity Game Engine-Official Site. *Online][Cited: October 9, 2008.] http://unity3d. com* ([n. d.]), 1534–4320.
[7] Michael Georgeff, Barney Pell, Martha Pollack, Milind Tambe, and Michael Wooldridge. 1998. The belief-desire-intention model of agency. In *International Workshop on Agent Theories, Architectures, and Languages*. Springer, 1–10.
[8] William Grant. [n. d.]. Extending the Q-Cog platform. ([n. d.]).
[9] Afsaneh Haddadi and Kurt Sundermeyer. 1996. Belief-desire-intention agent architectures. *Foundations of distributed artificial intelligence* (1996), 169–185.
[10] Mance E Harmon and Stephanie S Harmon. 1996. Reinforcement learning: A tutorial. *WL/AAFC, WPAFB Ohio* 45433 (1996).
[11] Sven Koenig and Reid Simmons. 1998. Xavier: A robot navigation architecture based on partially observable markov decision process models. *Artificial Intelligence Based Mobile Robotics: Case Studies of Successful Robot Systems* (1998), 91–122.
[12] Michael L Littman. 2009. A tutorial on partially observable Markov decision processes. *Journal of Mathematical Psychology* 53, 3 (2009), 119–125.
[13] R Andrew McCallum. 1995. Instance-based utile distinctions for reinforcement learning with hidden state. In *ICML*. 387–395.
[14] Waltham Michael. [n. d.]. Design and implementation of the Q-Cog Architecture. Unpublished manuscript. ([n. d.]).
[15] pemani4911. 2017. pemami4911/POMDPy: POMDPs in Python. (2017). https://github.com/pemami4911/POMDPy

[16] Lawrence Rabiner and B Juang. 1986. An introduction to hidden Markov models. *ieee assp magazine* 3, 1 (1986), 4–16.

[17] Guy Shani. 2007. *Learning and solving partially observable markov decision processes*. Ph.D. Dissertation. Ben-Gurion University of the Negev.

[18] Guy Shani, Ronen I Brafman, and Solomon E Shimony. 2005. Model-based online learning of POMDPs. In *European conference on machine learning*. Springer, 353–364.

[19] David Silver and Joel Veness. 2010. Monte-Carlo planning in large POMDPs. In *Advances in neural information processing systems*. 2164–2172.

[20] Richard S Sutton and Andrew G Barto. 2011. Reinforcement learning: An introduction. (2011).

[21] Svalorzen. 2017. Svalorzen/AI-Toolbox: A C++ framework for MDPs and POMDPs with Python bindings. (2017). https://github.com/Svalorzen/AI-Toolbox

[22] Unknown. 2017. pomdp-solve. (2017). http://www.pomdp.org/code/