

Q-Cog: A Q-Learning Based Cognitive Agent Architecture for Complex 3D Virtual Worlds

by

Michael Waltham

Submitted to the School of Mathematics, Statistics and Computer
Science

in partial fulfillment of the requirements for the degree of


Master of Science (Computer Science)


at the


UNIVERSITY OF KWAZULU-NATAL

September 2017

© University of KwaZulu-Natal 2017. All rights reserved.

Author 
School of Mathematics, Statistics and Computer Science
September 10, 2017


Certified by
Prof Deshendran Moodley
Associate Professor
Thesis Supervisor


Certified by
Mr Anban Pillay
Lecturer
Thesis Supervisor

Abstract

Intelligent cognitive agents should be able to autonomously gather new knowledge and learn from their own experiences in order to adapt to a changing environment. 3D virtual worlds provide complex environments in which autonomous software agents may learn and interact. In many applications within this domain, such as video games and virtual reality, the environment is partially observable and agents must make decisions and react in real-time. Due to the dynamic nature of virtual worlds, adaptability is of great importance for virtual agents. The Reinforcement Learning paradigm provides a mechanism for unsupervised learning that allows agents to learn from their own experiences in the environment. In particular, the Q-Learning algorithm allows agents to develop an optimal action-selection policy based on their experiences. This research explores the adaptability of cognitive architectures using Reinforcement Learning to construct and maintain a library of action-selection policies. The proposed cognitive architecture, Q-Cog, utilizes a policy selection mechanism to develop adaptable 3D virtual agents. Results from experimentation indicates that Q-Cog provides an effective basis for developing adaptive self-learning agents for 3D virtual worlds.

Acknowledgments

I, Michael Waltham, would like to firstly acknowledge the help provided by my supervisors Deshen Moodley and Anban Pillay. Secondly I would like to thank the Centre for Artificial Intelligence Research (CAIR) for funding me throughout this research.

Preface

The research contained in this dissertation was completed by the candidate while based in the Discipline of Computer Science, School of Mathematics, Statistics and Computer Science of the College of Agriculture, Engineering and Science, University of KwaZulu-Natal, Westville, South Africa. The research was financially supported by the Center for Artificial Intelligence Research (CAIR).

The contents of this work have not been submitted in any form to another university and, except where the work of others is acknowledged in the text, the results reported are due to investigations by the candidate.

Declaration: Plagiarism

I, Michael Waltham, declare that:

- (i) the research reported in this dissertation, except where otherwise indicated or acknowledged, is my original work;
- (ii) this dissertation has not been submitted in full or in part for any degree or examination to any other university;
- (iii) this dissertation does not contain other persons' data, pictures, graphs or other information, unless specifically acknowledged as being sourced from other persons;
- (iv) this dissertation does not contain other persons' writing, unless specifically acknowledged as being sourced from other researchers. Where other written sources have been quoted, then:
 - a) their words have been re-written but the general information attributed to them has been referenced;
 - b) where their exact words have been used, their writing has been placed inside quotation marks, and referenced;
- (v) where I have used material for which publications followed, I have indicated in detail my role in the work;
- (vi) this dissertation is primarily a collection of material, prepared by myself, published as journal articles or presented as a poster and oral presentations at conferences. In some cases, additional material has been included;

(vii) this dissertation does not contain text, graphics or tables copied and pasted from the Internet, unless specifically acknowledged, and the source being detailed in the dissertation and in the References sections.

Contents

1	Introduction	1
1.1	Background	1
1.1.1	Agent-Based Cognitive Architectures	1
1.1.2	Reinforcement Learning	2
1.1.3	Evaluating Agent Architectures in Virtual Worlds	4
1.2	Aims and Objectives	6
1.3	Impact and Contributions	6
1.4	Structure of the Dissertation	7
2	Literature Review	8
2.1	Agent Architectures	8
2.1.1	Belief, Desire, Intention	8
2.1.2	ACT-R (Adaptive Control of Thought-Rational)	9
2.1.3	SOAR (State, Operator and Result)	11
2.1.4	CLARION (Connectionist Learning with Adaptive Rule Induction ON-line)	14
2.1.5	Summary of Architectures	15
2.2	Reinforcement Learning and Policy Libraries	15
2.2.1	Temporal Difference Learning	16
2.2.2	Q-Learning Overview	16
2.2.3	Policy Libraries	17
2.3	Applications of Q-Learning in 2D-3D Virtual Worlds	18
2.3.1	Q-Learning for Autonomous Mobile Robot Navigation	18

2.3.2	A Q-Learning Based E-pet	20
2.3.3	Q-Learning for the Flocking of Agents	21
2.3.4	Deep Q-Learning in a Blocks World Environment	22
2.3.5	Deep Q-Learning in a First-Person Shooter Game	23
2.4	Agent Evaluation Methods and Environments	24
2.4.1	The Malmo Platform	24
2.4.2	Haunt 2 Navigation Agent	25
2.4.3	Predator-Prey Simulations	26
2.5	Summary of Literature Review	26
3	Design and Implementation of the Q-Cog Architecture	28
3.1	Architecture Objectives	28
3.2	Architecture Overview	29
3.2.1	Modules	30
3.2.2	The Q-Cog Decision Cycle	32
3.3	Policy Selection	33
3.4	Reference Q-Cog Implementation	36
3.4.1	Module Implementation	37
3.4.2	The Policy Selection Module	41
3.5	Summary	42
4	Design and Implementation of the Experimental Platform	43
4.1	Platform Requirements	43
4.2	The 3D Experimental Platform	44
4.2.1	The Generic Platform	45
4.2.2	The Simulation Engine	50
4.3	Platform Customization and Setup	51
4.3.1	Agent Development	51
4.3.2	Setting Up a Simulation	53
4.4	Summary	54

5	Experimental Design and Results	55
5.1	Scenario Overview	55
5.2	Experimental Platform Extension: Agent Combat	57
5.3	Experimental Design	58
5.3.1	Environment Design	58
5.3.2	Metrics	60
5.3.3	Setup and Execution of Simulation	60
5.4	Experiments and Results	61
5.4.1	Experimentation	63
5.5	Experimental Platform Evaluation	70
5.5.1	The Q-Cog Application	70
5.5.2	Other Applications	70
5.5.3	Experimental Platform Results Summary	73
5.6	Summary	73
6	Discussion	75
6.1	Analysis of the Q-Cog Architecture	75
6.1.1	System Design	75
6.2	The Policy Library	80
6.2.1	Design and Integration	80
6.2.2	The Policy Selection Process	81
6.2.3	Policy Selection in Game Environments	81
6.3	The Experimental Platform	82
7	Conclusion and Future Work	84
7.1	Future Work	86

List of Figures

2-1	An overview of the ACT-R cognitive architecture from [11].	10
2-2	An overview of the SOAR cognitive architecture	12
2-3	A depiction of the 2D robot navigation world utilized by Muhammad and Bucak [29].	19
2-4	A 3D model of the E-pet developed by Cheng et al [6].	20
2-5	Human interaction with the E-pet developed by Cheng et al [6].	20
2-6	A depiction of the 3D blocks world utilized by Barron et al [3].	22
2-7	A screenshot of the first-person shooter game Doom.	24
2-8	A screenshot within the Malmo platform from [19].	25
3-1	The Q-Cog agent architecture.	29
3-2	A diagram outlining the decision cycle of Q-Cog.	33
3-3	The situation-policy mapping within the Policy Selection Module.	35
3-4	The Policy Selection Module.	36
3-5	A UML diagram of the reference Q-Cog implementation.	37
4-1	An example simulation within the experimental platform.	44
4-2	A diagram providing an overview of entities within the experimental platform.	45
4-3	The agent’s vision.	46
4-4	A depiction of the navigation mesh in Unity3D. From: docs.unity3d.com/Manual/nav-BuildingNavMesh.html	48
4-5	A depiction of the TCP connection from the experimental platform to the agent architecture.	48

4-6	An overview of the experimental platform architecture.	49
4-7	An example of observation data transmission.	50
4-8	An overview of the simulation engine in the experimental platform.	50
4-9	A 3D humanoid model in the experimental platform.	52
4-10	The components of a humanoid agent within the experimental platform.	52
4-11	An example code snippet from the Entity Movement script.	53
5-1	An screenshot from a simulation in the experimental platform where two agents have engaged in combat.	58
5-2	A figure depicting a portion of a dataset containing simulation results.	61
5-3	Prefabs in the Unity engine allows various object types to be stored for later use and added to the scenario.	62
5-4	A depiction of the policy library containing the policies for predators A, B and C.	65
5-5	A figure depicting the average success of each agent utilizing the three parameter sets.	66
5-6	A figure depicting the average survival time of each agent utilizing the three parameter sets.	67
5-7	The single policy of the agent without policy selection in parameter set 3.	67
5-8	The developed policy for the agent against predator type A as stored in the policy library in parameter set 3.	68
5-9	The developed policy for the agent against predator type B as stored in the policy library in parameter set 3.	68
5-10	The developed policy for the agent against predator type C as stored in the policy library in parameter set 3.	69
5-11	Cumulative agent kills based on results obtained by Ikram [16].	71
5-12	A depiction of the hearing sensor as implemented by Ballim [2].	72
5-13	Results obtained by Ballim relating to agent object collection for different architecture implementations [2].	73

List of Tables

3.1	A table depicting an example section of a generated policy.	40
5.1	Reward values received as a result of each environment event.	58
5.2	A table representing the discretized state space.	59
5.3	Experimental Parameter Sets.	64
6.1	A comparison of cognitive architectures.	76

Chapter 1

Introduction

The goal of artificial general intelligence (AGI) is to develop agents that may adapt to different environments and are not specifically designed for a narrow task [32]. Cognitive agent architectures provide a framework to develop adaptive, intelligent agents that are capable of learning to act in their environment. These architectures are evaluated based on aspects such as agent learning capabilities and adaptability. There has been a recent interest in evaluating these autonomous agents within virtual 3D environments [45]. Virtual environments provide researchers with a real-time complex world that mimics real world properties. Researchers may also visualize agent behaviour which assists in design and evaluation.

1.1 Background

1.1.1 Agent-Based Cognitive Architectures

A cognitive architecture draws inspiration from theories and concepts in cognitive science to provide an architecture that models intelligent behaviour [11, 44, 27, 38]. An agent is an entity that is able to perceive its environment through sensors and act on the environment through actuators without human intervention [40, 48]. Agents possess knowledge about the environment that is either predefined by the developers or learned through their own experience. A cognitive architecture may therefore be

used to create a number of concrete, case-specific cognitive agents that are able to learn and understand the environment [38, 25, 21]. The architecture defines how agent knowledge is stored and processed in order to make decisions and achieve goals in the current environment [25]. A cognitive architecture should contain minimal initial data and structures and the agent should be able to learn from its own experiences in the environment [12].

1.1.2 Reinforcement Learning

An important aspect of an agent architecture is the learning mechanism. Reinforcement Learning (RL) is an unsupervised learning paradigm suitable for cognitive agents [1]. Reinforcement values are used by the agent to learn to make behavioural decisions. Unlike supervised learning paradigms, example data is not supplied to the algorithm to identify errors in decision making while training. An action selection policy may be generated from this reinforcement data allowing the agent to determine which action to select given each environment state. This policy determines which actions the agent should select given each environment state. It utilizes both exploration of new actions and the exploitation of its existing policy to identify the consequences of its actions while acting in the environment [15].

The purpose of a RL algorithm is to learn a policy, also known as a strategy, that can map environment states to optimal actions [49]. The agent follows this policy in order to effectively interact with the environment. Given a particular environment state, the agent may either choose to follow the action suggested by the existing policy or perform an unexplored action in an attempt to obtain better results for that state. Once an action is performed in a state, the agent may receive a reinforcement value as feedback which may be used to refine the policy. A clear limitation of conventional RL is that it is focused on learning a single task [9, 5]. There have been attempts to solve this issue with recent work investigating the development of multiple policies within a policy library in order for the agent to be able to adapt to a wider range of tasks in the environment [5]. This allows the agent to refine the current policy using

past learned policies as input.

The environment that the agent interacts with is generally represented by a Markov Decision Process (MDP) which contains a finite set of states and actions [49, 42]. During training, the agent performs a number of actions in different states and observes the outcome or reward from the environment. These reward values are then utilized to learn which actions are best to perform in each state. An RL algorithm aims at maximizing the expected reward and eventually generates an action selection policy that the agent may follow [13]. In many problem domains, such as game playing, Reinforcement Learning is the only applicable learning mechanism to create high quality intelligent agents as it is extremely difficult to provide accurate training data [36].

Markov Decision Process (MDP)

A MDP contains a set of states as well as a set of possible actions that may be performed in these states [13]. In a Markovian world, the result of an agent performing an action in the environment is dependant only on the current state of the world and the action performed [39]. In a MDP, it is assumed that the environment is fully observable. Formally, a MDP may be defined by the tuple (S, A, T, R) where [39]:

- S represents the finite set of world states.
- A represents the finite set of actions that may be performed by the agent in the environment.
- T represents the transition function defining the probabilities that an action a performed in a state s will result in state s' .
- R represents the reward function which defines state utilities and action execution costs. This function may define reward values for executing particular actions in particular states.

Policy Libraries

Recently, there has been an interest in moving away from the standard notion of a single policy generated by an RL algorithm to an approach where the agent maintains a library of policies [5, 9]. The agent is able to refine the current policy or select a more appropriate policy at each given time. This allows the agent to develop multiple policies for different aspects of the environment instead of attempting to develop a single general policy. The policy library enables agents to adapt to complex environments in which a single policy may not be sufficient for optimal decision making. A policy library system requires both a storage mechanism for generated policies and a policy selection mechanism to select the most appropriate policy. The policy selection mechanism need not add complexity to the agent and may be specifically designed for the problem domain.

1.1.3 Evaluating Agent Architectures in Virtual Worlds

Three-dimensional (3D) virtual worlds provide interesting environments for intelligent agents and are increasingly being used for evaluating performance of agents [19, 24, 46]. A virtual world will be defined as a computer generated three-dimensional space in which objects may interact with one another. Applications of virtual worlds include video games and virtual reality applications which attempt to create an immersive environment for the human user. Virtual 3D environments provide unique challenges for intelligent agents to overcome [19].

Virtual worlds have the following characteristics [19, 24]:

- Virtual worlds operate in real-time and so agents are forced to make decisions in real-time. This results in a limitation on available resources as agent logic competes with aspects such as graphics rendering and audio.
- Virtual worlds generally demand agents to solve multiple tasks simultaneously. For example agents may need to learn world navigation while simultaneously interacting with other agents in the environment.

- These environments are usually dynamic. Agents are required to adapt their behaviour when there are changes in the environment. Environment changes may render previous agent knowledge invalid and it is important for agents to change context in certain circumstances.
- Reinforcement Learning experimentation has largely taken place in 2D environments with full observability. 3D virtual environments are predominantly partially observable, which provides a far more realistic environment to evaluate agent performance.
- Virtual worlds containing multiple agents allow for AI-AI interactions.

The graphical quality of modern virtual worlds has become increasingly realistic over time [8]. This has stimulated a need for virtual agents to be able to match the immersion level provided by computer graphics in applications such as games and virtual reality applications. Unsupervised learning paradigms provide the necessary support to allow agents to learn about an environment without necessarily adjusting underlying agent logic. Reinforcement Learning in particular allows the agent to develop a self-learned action selection policy without needing modify pre-defined behaviours.

Agent Learning in Virtual Worlds

Virtual worlds contain a number of complexities, such as large state spaces and dynamic natures, that make agent learning difficult. Reinforcement Learning allows agents to utilize their own experiences in the environment to attempt to develop an optimal action selection policy. Supervised learning mechanisms may not allow the agent to develop a complete model of the environment as the training dataset may be incomplete.

The exploration process for an agent within a virtual world may never end as new regions of the world are discovered and new unforeseen circumstances are encountered. Agent learning within these worlds usually involves unsupervised learning methods

such as Reinforcement Learning and the agent may need to constantly adjust its policy within the environment in a similar way to a human [20].

1.2 Aims and Objectives

This research explores the effectiveness of Reinforcement Learning with a policy selection mechanism as the primary learning mechanism in a cognitive agent architecture. The aim of this research is to design and evaluate a Q-Learning based cognitive agent architecture that may be used to create intelligent, adaptive, self-learning agents within the 3D virtual world domain. In particular, this architecture must provide the required mechanisms for the agent to autonomously develop multiple policies to adapt to changing aspects in the environment. The following outlines the objectives of this research:

- Develop a virtual 3D experimental platform that may be reused in future agent-based research.
- Design a cognitive agent architecture that allows 3D virtual agents to adapt to a changing environment using concepts from state-of-the-art architectures.
- Integrate a policy library into the architecture to allow agents to adapt to drastic environment changes.
- Evaluate the architecture using a complex scenario designed and implemented within the experimental platform.

1.3 Impact and Contributions

The focus of the work is an architecture for developing intelligent and adaptive agents in the 3D virtual world domain. The architecture provides a framework allowing agents to adapt behaviour based on changes in the environment by utilizing a policy selection mechanism. The performance of agents utilizing a cognitive architecture

and Q-Learning with multiple policies is empirically evaluated within a 3D virtual environment.

A reusable 3D experimental platform is built for the evaluation. The platform is generic and reusable and has room for custom extensions. The platform was designed to allow for ease of integration with new architectures and algorithms.

The outcomes of the work may be applied to many different fields including game development and AI research. A major application of agent architectures in 3D virtual worlds is game applications. Game developers constantly seek to improve the realism and unpredictability of their agents. The architecture proposed in this work provides an effective method of developing adaptive agents that may learn from their environment. In the domain of game development, agents can gain insight into the behavioural patterns of human players and develop interesting policies to challenge the players. Researchers and educators may utilize the experimental platform to evaluate and explain agent based systems. The extensible nature of the platform provides the opportunity to explore many aspects of agent based research within virtual 3D environments.

1.4 Structure of the Dissertation

The remainder of this dissertation is structured as follows: Chapter 2 introduces previous related work. Details of the proposed architecture is then described in chapter 3. Chapter 4 describes the implementation of the experimental platform. Chapter 5 then outlines experiments performed in order to evaluate the architecture and analyses the results obtained from each experiment. Chapter 6 provides a discussion on topics relating to the architecture and the experimental platform. Lastly, Chapter 7 provides concluding remarks and proposes future work.

Chapter 2

Literature Review

This chapter introduces previous work in the field. The chapter firstly introduces various well known agent architectures and cognitive agent architectures. Secondly, previous work regarding Reinforcement Learning and policy libraries is mentioned. Thirdly, various case studies involving Q-Learning agents in virtual worlds are described. Lastly, previous work involving the evaluation of agent architectures is discussed.

2.1 Agent Architectures

This section introduces well known agent architectures and cognitive architectures found in literature. This includes the Belief, Desire, Intention (BDI) framework and the SOAR, ACT-R and CLARION cognitive architectures.

2.1.1 Belief, Desire, Intention

The BDI framework has been used extensively for intelligent cognitive agents [34, 40]. A BDI agent carries out complex behaviours through the use of a goal-plan structure [40]. Agents are able to simultaneously manage several goals [35]. Each agent contains a set of beliefs which forms the current belief state of the agent. The set of desires indicates the goals that the agent is aiming to achieve. The set of intentions

indicate the desires that the agent has decided to act on [4]. The BDI framework provides a balance between goal planning and reactive behaviour. A standard BDI implementation is, however, unable to deal with information that is not complete as a result of sensors that may be unreliable or noisy [4].

2.1.2 ACT-R (Adaptive Control of Thought-Rational)

ACT-R is a widely used cognitive architecture focused on modelling human cognition [18, 11, 25, 7]. It contains different modules, each of which focus on handling different types of information.

Overview

A focus of the ACT-R architecture is modularity [12]. Each module processes information independently of other modules and may communicate through buffers [18]. This allows each module to run in parallel without interfering with the processing of another module. The modules in ACT-R include the sensory, action, intentional and declarative modules and have been associated with different regions of the human brain [11]. Each module has its own buffer which collectively makes up the short-term memory of the architecture. Each buffer may store up to one *chunk* which is a collection of name-value pairs [11]. Standard modules in ACT-R are:

- The Procedural module is the central module in the architecture. Production rules are present here and may activate to output an action when a particular condition is met.
- The Manual module performs manual actions such as pressing keys and moving the mouse.
- The Goal module provides context for the actions of agents. It allows the current goal to be broken up into smaller sub-goals that the agent may achieve. The current goal is stored within the goal buffer that is accessible to production rules.

- The Declarative module contains facts about the world that other modules may utilize.
- The Visual module observes and encodes all visual data.
- The Vocal module converts strings into speech.
- The Aural module observes and encodes audio data.
- The Imaginal module temporarily stores incoming perceptions.

All modules may communicate with each other via the Procedural module, as illustrated in Figure 2-1 taken from [11].

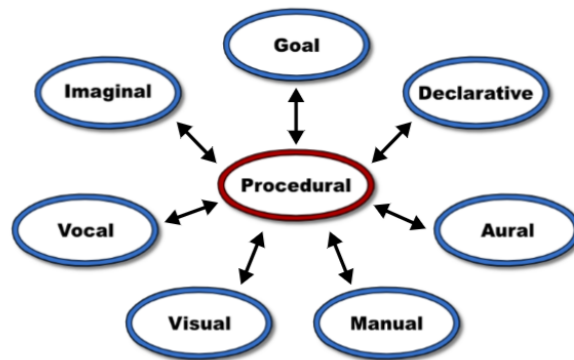


Figure 2-1: An overview of the ACT-R cognitive architecture from [11].

ACT-R stores both declarative and procedural knowledge [37]. Declarative knowledge represents perceived or factual information, for example the agent may store the location of a 3D object within a virtual world. Procedural knowledge consists of condition-action rules that the agent follows. When the condition of a production rule is satisfied, the corresponding action may be executed.

Learning and Action Selection

ACT-R uses utility values to determine which action is the most appropriate to execute given the current environment state [11]. This utility value is calculated using the difference between the estimated cost and benefit of each production rule. The

production rule with the highest expected utility is selected and executed.

ACT-R utilizes production compilation and rule utility [12]. Production compilation involves creating new production rules to replace multiple existing rules which reduces execution time in future cycles. The utility values are used to determine the best action to select at a given time. As an action is performed, the success and cost is re-evaluated [25].

The ACT-R architecture has been used in human-robotic interaction simulations [25]. Certain video game developers have previously utilized the ACT-R architecture when developing virtual agents [37].

2.1.3 SOAR (State, Operator and Result)

Overview

SOAR was one of the first cognitive architectures to be proposed [7]. The architecture focuses on the idea of problem solving. It begins at an initial state with the aim of reaching one of the goal states by selecting the best available operator at each update [12, 23]. An operator is applied to the current state to produce a new state [23]. The architecture is symbolic as symbols are used to represent objects [27]. Figure 2-2 depicts an overview of the architecture taken from [12].

Information about the current state, including both sensed environment data and internal inferences, are stored in working memory elements which represent the short term memory [12, 22, 31]. These working memory elements are attribute-value pairs. Working memory additionally contains information relating to the current goal.

The long term memory consists of procedural, semantic and episodic memory as illustrated by Figure 2-2. Knowledge in long term memory is structured as sets of production rules [12, 31]. When the conditions specified by a production rule are satisfied based on the contents of working memory, the production rule fires its

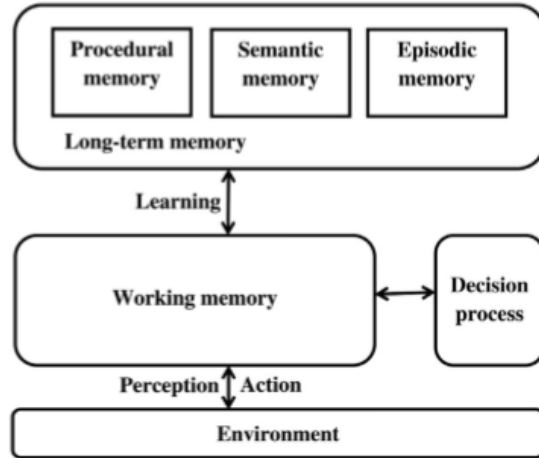


Figure 2-2: An overview of the SOAR cognitive architecture

action [31]. This action may modify the contents of working memory. Modifications to working memory may result in actions being performed on the environment which may then result in additional production rules firing to create a continuous cycle. A unique aspect of the SOAR architecture is that it allows all matching production rules to fire in parallel [31].

Learning and Action Selection

The current problem is represented internally by the current state in working memory [27, 23]. During the action selection process, when the condition of a production rule is met based on the contents of working memory, the operator specified by that rule is selected for execution [31]. This operator is applied to the current state in order to produce a resulting state [31]. In a similar way to ACT-R, SOAR uses utility values to select an operator with the highest expected performance given the current state [27, 12, 22]. When two operators are tied for selection, the current goal of the architecture is changed to the task of determining which operator to select [27, 25]. A more detailed description of the decision cycle of SOAR, taken from [12], is as follows:

1. Update working memory with input from sensors.
2. Execute all rules with conditions that match working memory. The results of these rules firing are placed into working memory. This stage ends once no new

knowledge has been added to working memory and it is stable.

3. Production rules propose applicable operators based on their conditions and the contents of working memory. Rule conditions ensure that operators are only proposed when relevant.
4. Select an operator by evaluating the operators proposed in (3) with respect to the contents of working memory.
5. Production rules that match the selected operator make changes to the current state. This includes the creation of motor commands.
6. Motor commands are processed.

Originally, the focus of SOAR was to build its knowledge base using predefined expert knowledge [12]. A process known as *chunking* is now used during learning [27, 25]. Chunking involves adding new rules to long-term knowledge using data in short-term memory [22]. The chunking module in SOAR activates whenever an *impasse* occurs. When the current knowledge is not sufficient to reach the desired goal, an impasse is reached [12]. When an impasse is resolved, chunking allows the system to generate a new rule (now referred to as a chunk) that will prevent a future impasse in this situation. The chunk is generated by converting the collected working memory elements during the impasse into conditions and actions of production rules [27].

SOAR-RL

The SOAR architecture largely uses a symbolic knowledge representation [30]. The SOAR-RL modification includes the addition of a Reinforcement Learning aspect which provides numeric knowledge representations. The extension allows agents to identify the expected rewards of performing actions. Reward values are received as additional inputs from the environment.

Originally, operator selection in SOAR was performed based on symbolic preference values. The addition of Reinforcement Learning allows the consideration of numeric preferences in the case where symbolic preferences do not provide enough information in order to make a decision.

The SOAR architecture has been applied to virtual environments such as the Haunt 2 game environment. A SOAR agent was developed by Helie and Sun [12] to navigate the environment and find an object of interest.

2.1.4 CLARION (Connectionist Learning with Adaptive Rule Induction ON-line)

Overview

The CLARION architecture allows agents to function with very little prior knowledge [12, 21, 27]. CLARION is divided into four main subsystems: The action-centered subsystem (ACS), the non-action-centered subsystem (NACS), the motivational subsystem (MS) and the meta-cognitive subsystem (MCS). Each subsystem is divided into a top and bottom level. The top level encodes explicit knowledge and the bottom level encodes implicit knowledge.

- The ACS controls all agent actions. These actions may be external such as motor commands, or internal such as knowledge reasoning. The bottom level of the ACS allows CLARION to adapt to situations that may not necessarily be solved using basic top level rules. It is made up of many connectionist networks which each handle a different type of input or task. The top level contains explicit symbolic rules.
- The NACS contains knowledge about the world in the form of semantic and episodic memory. The knowledge stored in both the top and bottom levels are non-action-centered. Actions performed by this subsystem include memory retrieval and inference.

- The MS maintains agent goals. It autonomously selects and focuses on important environment aspects. It contains both explicit goals and implicit drives.
- The MCS provides feedback to other modules regarding actions performed as a form of Reinforcement Learning. It may also guide and control the processing of other modules to improve performance.

Top-Down and Bottom-Up Learning

Within each module, CLARION uses a combination of bottom-up and top-down learning [21, 27, 12]. Bottom-up learning involves the conversion of implicit knowledge into explicit knowledge which is stored at the top level. Top-down learning involves the conversion of explicit knowledge into implicit knowledge which is stored at the bottom level.

2.1.5 Summary of Architectures

The focus of this section was aimed at reviewing commonly used cognitive agent architectures. The three discussed cognitive architectures: SOAR, ACT-R and CLARION contain many fundamental aspects of developing intelligent agents. Apart from the structural differences in these architectures, each architecture uses different learning techniques. These learning techniques include: Production compilation and rule utility in ACT-R, chunking in SOAR and top-down and bottom-up learning in CLARION. The SOAR architecture has also recently integrated a Reinforcement Learning module known as SOAR-RL in order to provide numeric knowledge representation.

2.2 Reinforcement Learning and Policy Libraries

This section introduces the concept of Reinforcement Learning and policy libraries. A policy may be viewed as a rule that agents may follow in order to select actions for each environment state. It is a mapping from environment states to actions

[41]. Temporal difference methods and the Q-Learning algorithm are described and previous work relating to policy libraries is discussed.

2.2.1 Temporal Difference Learning

There are many ways for an agent to learn a policy. Temporal difference (TD) is a Reinforcement Learning method that uses reward values to generate an action-value function [42]. The action with the highest value at each state is selected for execution. This method of learning provides a method of foreseeing the effects of an action in order to build an optimal policy [26].

2.2.2 Q-Learning Overview

Q-Learning is a model-free Reinforcement Learning algorithm that assigns reward values to state-action pairs [47]. When action a is performed in state s to produce a resulting state s' , a reward value is received and used to calculate the corresponding Q-Value for the state-action pair (s, a) using Equation 2.1 [28]. By learning the outcomes of actions through reinforcement values, Equation 2.1 allows the agent to develop expected utilities for executing an action a in a state s . The action selection process then involves maximizing the expected utility by selecting the action with the highest Q-Value in each environment state. The learning process, similar to that of Temporal Difference Learning [47], is to execute an action in a given state and evaluate the consequences based on reward feedback received.

The Q-Learning algorithm has been proved to eventually converge to the optimal policy within any finite MDP [15, 14]. A disadvantage of the algorithm is that the convergence rate varies based on the complexity of the environment [29]. Traditional Q-Learning only modifies a single Q-Table entry at a time and so with larger state spaces in complex environments, such as virtual 3D worlds, convergence may be significantly slower than in other environments.

$$Q(s_{t+1}, a_{t+1}) = Q(s_t, a_t) + \alpha \cdot [\Gamma + \gamma \cdot \max_{a' \in A} Q(s', a') - Q(s_t, a_t)] \quad (2.1)$$

Equation 2.1 represents the standard Q-Learning algorithm where: α represents the learning rate of the algorithm, Γ is the reward value and γ is the discount factor. The value of γ represents the extent to which the algorithm considers future rewards. A γ value of zero indicates that only immediate reward feedback is considered whereas a value of one indicates that future rewards are greatly considered. $\max_{a' \in A} Q(s', a')$ represents the maximum possible Q-Value that may be obtained given the next state s' and all actions a' .

After each interaction with the environment, the value function of the current state-action pair is updated using Equation 2.1 [33]. The Q-Learning algorithm selects the action with the highest Q-Value corresponding to the current state. Q-Values may be stored in a tabular structure referred to as a Q-Table.

2.2.3 Policy Libraries

The process of policy reuse involves the storing of learned policies for later reuse [9]. A library is maintained by the agent in order to store and retrieve a number of policies.

Initial work by Fernandez and Veloso [9] investigated effective ways of developing a policy library. A similarity metric was proposed to identify new policies that should be added to the library. If a new policy was substantially unique, it was added to the library. The proposed method of library formation was evaluated within a 2D robot navigation environment. The robot was aware of its location in the world and was able to move in four directions to reach the goal state. Their experiments obtained promising results and showed a 100% increase in performance due to policy reuse.

Recent work by Chalmers et al. [5] incorporated ideas explored by Fernandez and Veloso. This work explored ways to improve the adaptation times of agents when re-

acting to changes in the environment. They proposed the use of transfer learning to do this in order to utilize previously learned information. Abstraction of state spaces involves removing state variables that are not necessarily required in certain situations. Chalmers et al. [5] proposed a method of combining a state-space hierarchy with a policy storage and retrieval system. The hierarchical state space allowed them to adapt previous policies to recent problems in the environment. Context switching was also incorporated, where the agent is able to store recent state transitions and select a policy from memory that best predicted these transitions. Chalmers et al. [5] evaluated the work within a 2D robot navigation environment where the goal of the agent remained constant. The results of experiments where the agent utilized the extensions done by Chalmers et al. [5] are compared to that of an agent that utilized standard RL algorithms. It was found that the integration of their hierarchy-based adaptation allowed the agent to learn certain policies faster. The agent implementing standard RL was unable to compete as it was not able to adapt fast enough to changes in the environment.

Policy reuse allows the architecture to store a new policy in memory to be utilized and modified later. Collectively, these policies form a library in memory. Previous work done in this area showed promising results but evaluation was limited to 2D virtual environments.

2.3 Applications of Q-Learning in 2D-3D Virtual Worlds

The following section aims to outline state-of-the-art applications of the Q-Learning algorithm.

2.3.1 Q-Learning for Autonomous Mobile Robot Navigation

Muhammad and Bucak [29] proposed a modification of the Q-Learning algorithm by improving the rate at which the Q-Table of an agent converges to optimal values. Traditionally, Q-Learning modifies only a single Q-Table value per training episode. In

large state spaces the convergence rate becomes impractical. The algorithm proposed involves state-clustering in order to rearrange the state-action trajectory into non-repeated loops of states and actions [29]. It was noted that it is important for the agent, in this case a robot, to be fully autonomous in the domain of robot navigation. RL was chosen as it allows agents to modify a policy given their own experiences in the environment. The domain was chosen due to a high number of discrete states and possible actions.

The environment is a 2D world in which a robot must find its way to the goal state while avoiding obstacles. Figure 2-3 provides an illustration of the environment. The agent is able to observe both its position in the world and obstacles in its path. The agent may move left, right, up or down by either 1, 3, 10 or 15 pixels resulting in a total of 16 possible actions that may be executed.

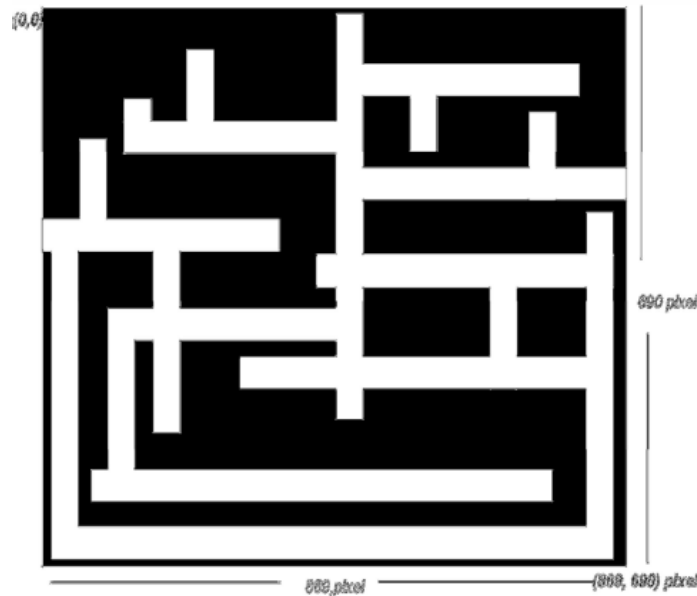


Figure 2-3: A depiction of the 2D robot navigation world utilized by Muhammad and Bucak [29].

A simulation was executed 1400 times. The rate of change in Q-Values was recorded and compared to that of traditional Q-Learning. The proposed algorithm greatly improved the rate of change in Q-Values. The convergence rate of the proposed algorithm was found to improve on traditional Q-Learning.

2.3.2 A Q-Learning Based E-pet

An E-pet may be either a virtual or robot animal companion. Cheng et al [6] proposed a Reinforcement Learning based framework for controlling the behaviour of an intelligent E-pet. The Q-Learning algorithm was utilized along with a lookup table containing the corresponding Q-Values. The framework aimed to utilize a planning scheme to reach a desired goal state.

The E-pet was implemented as a virtual 3D agent which interacted with human users as depicted in Figure 2-4. The agent had to select actions to perform in the environment based on instructions given by human users and reward signals received.



Figure 2-4: A 3D model of the E-pet developed by Cheng et al [6].

Users interacted with the E-pet by issuing commands using an interface depicted in Figure 2-5. After each interaction, the agent updates a Q-Table given the reward value issued by the user. This reward value is issued based on the extent to which the E-pet correctly performed the requested task.



Figure 2-5: Human interaction with the E-pet developed by Cheng et al [6].

The agent was trained using seven commands from the user. These commands

were: *Circle, run, up, sit, down, lie* and *come*. The agent was trained for a total of 60 iterations per instruction. Each action was evaluated according to how many iterations it took to reach the goal state. Results indicate that the number of planning states in their proposed framework is reduced by Q-Table updates.

2.3.3 Q-Learning for the Flocking of Agents

Hung et al [13] proposed a framework, known as Q-flocking, based on the Q-Learning algorithm for the flocking behaviour of UAVs. There has been an interest in investigating the deployment of multiple unmanned aerial vehicles (UAVs) that may collaborate for both military and civilian applications. Intelligent UAVs may be applied in areas such as rescue missions and agriculture. An evident solution, and the solution that has been utilized in previous work, is flocking and swarming AI algorithms. A complete model of the environment is often not available to the agent and so there has been investigation into autonomous robots that learn through their own experiences as opposed to having pre-defined models.

Q-flocking was evaluated using UAVs that learned flocking behaviours within a stochastic environment. Two experiments were performed and results were compared to related work by testing and comparing the performances of generated policies. The goal of the first experiment was to evaluate if Q-flocking was viable to allow agents to learn a flocking behaviour policy within a stochastic environment. Results indicated that agents were able to develop a policy allowing flocking behaviour involving a single leader. It was shown that the Q-flocking framework allows agents to find a near optimal policy given sufficient time. The goal of the second experiment was to verify that agents could adapt previous policies to a new environment. The results of the first experiment were used as the benchmark for the new results obtained. Results indicated that agents were able to successfully adapt their policies to the new environment, indicating that Q-flocking works effectively in stochastic non-stationary environments.

2.3.4 Deep Q-Learning in a Blocks World Environment

Barron et al [3] investigated agent performance while visually processing 3D virtual worlds and trained using Deep Q-Learning Networks. Evaluation consisted of a 3D blocks world similar to that developed by Johnson et al [19]. Figure 2-6 depicts a scene in the blocks world used.

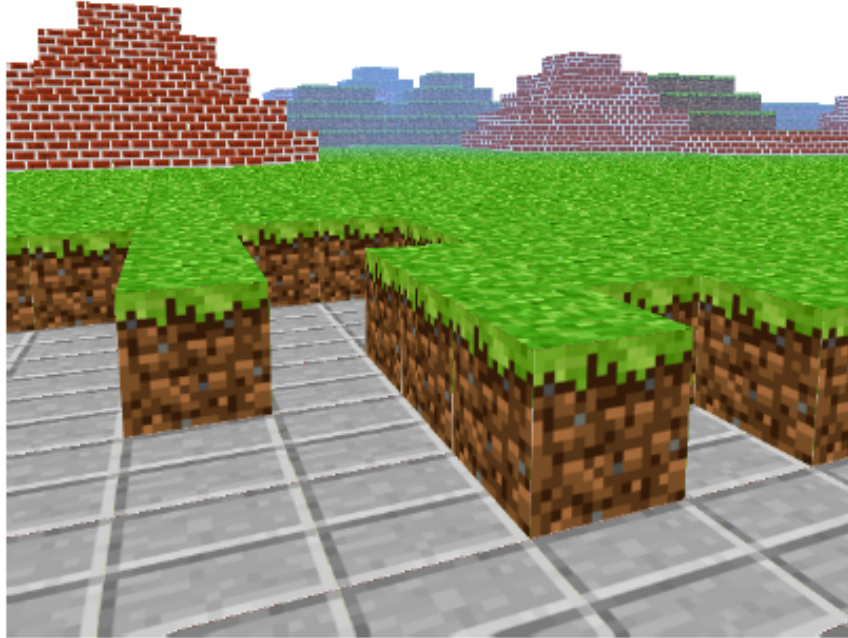


Figure 2-6: A depiction of the 3D blocks world utilized by Barron et al [3].

The agent viewed the world from a first-person perspective. It received 84x84 pixel screenshots of the world at each time step. The discrete actions available to the agent were as follows:

- Rotate the camera view up, down, left or right.
- Step forward, backwards, left or right.
- Jump forward.
- Attempt to break a block within the world.

The first set of experiments involved evaluating how effectively the agent may determine the distances to visible objects in the environment. This is important

given the first-person perspective as the agent is acting within a 3D world. Results indicated that the networks can determine distances to a relatively high degree of accuracy.

The second set of experiments presented a number of tasks for the agent to complete within the world. These tasks involved breaking the correct block types and avoiding certain blocks while navigating through the world. The complexity of the world was increased over a number of scenarios. Both shallow and deep networks were evaluated in these tasks and it was found that the shallow network performed better in the simple tasks.

2.3.5 Deep Q-Learning in a First-Person Shooter Game

Lample and Chaplot [24] proposed an architecture to be used within first person shooter games. Both Deep Q-Networks and Deep Recurrent Q-Networks were utilized within the architecture. They note that previous applications of these algorithms were within 2D games that do not effectively represent real world properties. They evaluated their proposed architecture within the 3D first-person shooter game Doom depicted in Figure 2-7.

Lample and Chaplot noted certain unique challenges that 3D environments provide for agents to solve. These challenges include:

- Navigation through a complex 3D terrain.
- Item collection.
- Enemy recognition and combat simulations.
- States are usually partially observable.

The game world utilized provided the agent with a first-person view allowing the work to be closely related to robotics applications.

The agents were evaluated by firstly being trained utilizing Lample and Chaplot's [24] architecture and evaluated against the standard built-in game agents. Simulations



Figure 2-7: A screenshot of the first-person shooter game Doom.

were performed in both known and unknown environments and metrics such as the number of agent kills were recorded. Agents were secondly trained and evaluated against a number of human players. Their results indicated that the agent was able to perform better than both the standard game agents and the human players used.

2.4 Agent Evaluation Methods and Environments

This section provides scenarios and environments that have been used to evaluate agents.

2.4.1 The Malmo Platform

An artificial intelligence testing platform known as Malmo has recently been released by Johnson et al [19], an example scene is illustrated in Figure 2-8. This platform was built on an exploration based video game known as Minecraft. The platform provides

an interface for creating agents within a virtual 3D game environment. The main goal of the platform is to assist in research involving Artificial General Intelligence; in particular, the platform seeks to provide assistance when experimenting with agents that should be flexible in nature and adapt to their surroundings. Johnson et al noted that an efficient means of RL experimentation is one of the contributions of the platform [19].



Figure 2-8: A screenshot within the Malmo platform from [19].

2.4.2 Haunt 2 Navigation Agent

A SOAR agent was developed to navigate through the Haunt 2 game environment [12]. The purpose of the agent was to autonomously learn the subgoals and operations required to find an item of interest in the environment. Experimentation aimed to verify that agents were able to learn to navigate the environment. The agent was trained using input from another hard-coded SOAR agent that had previously run the simulation demonstrating that one SOAR agent may be trained using the knowledge of another.

These virtual environments provide complex scenarios that challenge the adaptive learning capabilities and highlight the importance of realistic environments for evaluating virtual self-learning agents.

2.4.3 Predator-Prey Simulations

A predator-prey simulation was performed by Kazemifard et al [21] while investigating the processing of emotion [21]. The goal of the prey is to survive as long as possible. The three agent types are predators, prey and grass. The predators are not able to be killed and their population increased over time. The population of the prey is also able to grow however, the grass population slowly decreases. This stresses the importance of an adaptive prey, as the prey with the best plan would survive the longest.

A similar predator-prey simulation was performed by O. Javier and R. Lopez [17] involving an environment known as Animat. Prey in the environment compete for food and water sources while attempting to avoid predators.

2.5 Summary of Literature Review

The goals of this literature review were as follows:

- Identify state-of-the-art agent architectures used when developing intelligent and autonomous agents.
- Analyse previous case studies involving Reinforcement Learning based autonomous agents in virtual world environments.
- Identify realistic methods and environments for agent evaluation.

Three commonly used cognitive agent architectures: SOAR, CLARION and ACT-R were identified and examined. These architectures surpass simple behaviour control techniques by giving agents the capability to learn and adapt to their environment. In particular, recent additions to the SOAR architecture includes a Reinforcement Learning module. This allows reinforcement values from the environment to be utilized in the SOAR decision cycle. Although this allows agent experiences in the environment to effect the outcome of decisions, the basis of the architecture is not centred around Reinforcement Learning. An advantage of centering an architecture

around Reinforcement Learning is that it allows the agent to learn purely based on its own experiences in the environment as opposed to requiring a certain amount of pre-inserted expert knowledge. This may be particularly important in game environments similar to the one utilized by Lample and Chaplot [24] where the agent competes against human players with largely stochastic behaviour. The outputs of such an architecture is one or more strategies that the agent may follow to interact optimally within the environment.

In order to allow agents to learn from their own experiences in the environment, the Q-Learning algorithm was explored as well as a policy library extension. It is evident that the policy library extension has greatly improved agent performance in previous work performed. Other applications of the Q-Learning algorithm within 2D and 3D virtual worlds were also explored.

The review also highlighted the need for complex, realistic virtual environments in order to evaluate agent architectures. A great deal of previous work has been performed using 2D environments that do not provide the level of complexity and realism that 3D environments contain.

Chapter 3

Design and Implementation of the Q-Cog Architecture

This research explores the effectiveness of Reinforcement Learning with a policy selection mechanism as the primary learning mechanism in a cognitive agent architecture. This chapter describes the design and implementation of Q-Cog, a cognitive agent architecture for adaptive self-learning agents in complex and uncertain virtual 3D environments. The architecture allows agents to gather knowledge about the environment through their own experience and thus an important feature of the architecture is a Reinforcement Learning mechanism. The mechanism is enhanced with a dynamic policy selection mechanism that allows agents to adapt effectively to new and previously unknown situations in the environment.

3.1 Architecture Objectives

The purpose of the architecture is to allow the development of adaptive self-learning agents in 3D virtual environments. The requirements of the architecture are:

- Provide a dynamic policy generation and selection mechanism to allow agents to learn and adapt effectively to changes in the environment. Policies must be automatically generated and refined by the architecture during runtime.

- Provide a mechanism for generating perceptions from environment observations (defined below in Section 3.2). This allows complex 3D environment observations, such as the 3D world locations of other agents to be simplified into a set of perceptions that may be utilized during learning and action selection.

The Q-Cog architecture is centered around 3D virtual agents. Agent sensors generate observations which require interpretation. These observations are interpreted and discretized into a set of perceptions that are far easier to process in later steps of the architecture’s decision cycle.

3.2 Architecture Overview

Certain components of the architecture, such as the central processing and memory components, were inspired by the SOAR and ACT-R cognitive architectures, while components that form the policy selection mechanism were uniquely crafted in order to satisfy the objectives. The proposed architecture is shown in Figure 3-1 and each component is described in detail thereafter.

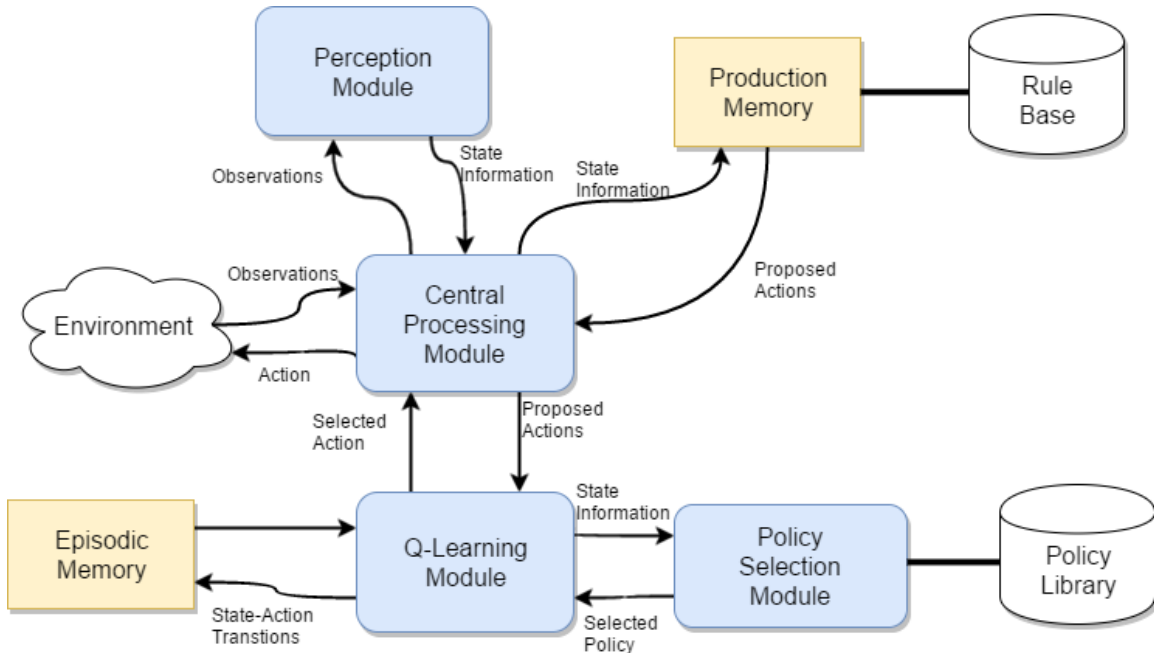


Figure 3-1: The Q-Cog agent architecture.

3.2.1 Modules

The Central Processing Module

The Central Processing Module controls the flow of execution by controlling the input and output of other modules. The module calls appropriate functions within other modules in order to execute the main event loop of the architecture. As depicted in the architecture overview (Figure 3-1), actions proposed for execution from Production Memory are firstly retrieved by the Central Processing Module before being passed as input to the Q-Learning Module. This allows tight control over the flow of information and module execution order at each world update. Additionally, it is the only module that interacts with the environment by receiving observations and performing selected actions at each timestep.

The Perception Module

Many observations from the world are continuous values, such as locations in the 3D space or terrain navigation data. This results in an infinite number of possible world states. The Perception Module discretizes observations into perceptions by mapping continuous world space data into a set of discrete values. For example, the module may translate world locations into discrete distance representations of *near* and *far*.

The Production Memory Module

Production memory contains information relating to actions the agent may perform in the environment. Production rules of the form: $(Conditions) \rightarrow Propose(Action)$ specify the conditions that determine when an action is proposed for execution. Each production rule defines the conditions for a single action to be proposed. Perception data may cause a number of these production rule conditions to be true and thus propose the associated actions.

Firstly, the module receives perceptions generated by the perception module. When the condition(s) of a particular action is met based on the current perceptions, that corresponding production rule will fire and the action will be proposed.

For e.g: ($PredatorCount > 3$) \rightarrow $Propose(Flee)$ relates to a humanoid agent surviving in an environment containing a number of predators. This example production rule proposes the action $Flee$ when the number of perceived predators is greater than three. In this example, the Perception Module would firstly output the value of $PredatorCount$ based on raw observations in the environment. There may be additional variables contributing to state information. The Production Memory Module would then insert this value into the rule engine to infer proposed actions. In the case that the value of $PredatorCount$ exceeds three, the rule engine would output the $Flee$ action along with other actions that are appropriate for the current state. Once all state information from the Perception Module has been inserted into the rule engine, a set of proposed actions is generated and fed through to the Central Processing Module. The Q-Learning module will then only consider the proposed action subset when selecting an action.

The Q-Learning Module

The Q-Learning Module aims to produce and refine a number of policies for the agent to follow to allow it to act effectively and efficiently within the environment. At the lowest level, the Q-Learning algorithm is used to refine a policy during learning. The module takes as input the proposed actions for this decision cycle and outputs a selected action from the proposed actions which may be optimal depending on the amount of training the agent has received. Reinforcement values are also received from the environment and are utilized by the Q-Learning algorithm during policy refinement. The Q-Learning Module refines a selected policy at each timestep. This selected policy is then used by the module in order to select an action to perform. All generated policies are then stored in the Policy Selection Module for later use.

The Episodic Memory Module

After an action is performed in a particular state, the mapping to the resulting state is stored. A function defined by $F(s, a|s')$, which maps a state-action pair (s, a) to a resulting state s' will eventually be formulated using this stored data. The Q-Value

calculation (Equation 2.1) has therefore been adapted to include the data stored in episodic memory. The adapted equation 3.1 utilizes the formulated function which maps state-action pairs to resultant states. All possible resulting states are used in the equation as a single resulting state may not always be guaranteed in stochastic environments. The agent is therefore able to learn $P(s'|s, a)$, which is the probability of reaching state s' given that action a was performed in state s , by making use of episodic memory. The agent is therefore able to learn a state transition model for the environment. This information may then be shared across multiple agents in order to increase the learning rate or may be passed to future agents to be utilized in other simulations.

$$Q(s_{t+1}, a_{t+1}) = Q(s_t, a_t) + \alpha \cdot [\Gamma + \gamma \cdot \max_{s' \in S, a' \in A} Q(s', a') - Q(s_t, a_t)] \quad (3.1)$$

In the above equation, α represents the learning rate of the algorithm, Γ is the received reward value and γ is the discount factor. The value of γ represents the extent to which the algorithm considers future rewards. A γ value of zero indicates that only immediate reward feedback is considered whereas a value of one indicates that future rewards are weighted more. The change from equation 2.1 involves $\max_{s' \in S, a' \in A}$ whereby each possible resulting state is visited when calculating the maximum Q-Value as opposed to the standard Q-Learning algorithm where a single resulting state is assumed. This change was introduced due to the stochastic nature of virtual environments. A definitive resulting state is not always guaranteed and so all possible resulting states must be taken into account when refining Q-Table values.

3.2.2 The Q-Cog Decision Cycle

The decision cycle of the architecture is the main event loop that defines information flow and decision points. The decision cycle of Q-Cog in Figure 3-1 can be generalized into 6 main steps and is outlined in Figure 3-2.

1. Observations from the environment are stored in the Central Processing Module.

2. The observations from the Central Processing Module are passed to the Perception Module. The Perception Module then processes these observations into perceptions.
3. Production Memory retrieves the perceptions generated by the Perception Module from the Central Processing Module. These perceptions are then used as input to a rule engine to infer possible actions for the agent.
4. Proposed actions along with the contents of Episodic Memory are taken as input by the Q-Learning Module for use during action selection.
5. State information, that is, the collection of perceptions, is then given to the Policy Selection Module and a policy is selected from memory for execution.
6. Using the selected policy along with the proposed actions, the Q-Learning Module selects an action to perform and returns it to the Central Processing Module which execute the action.

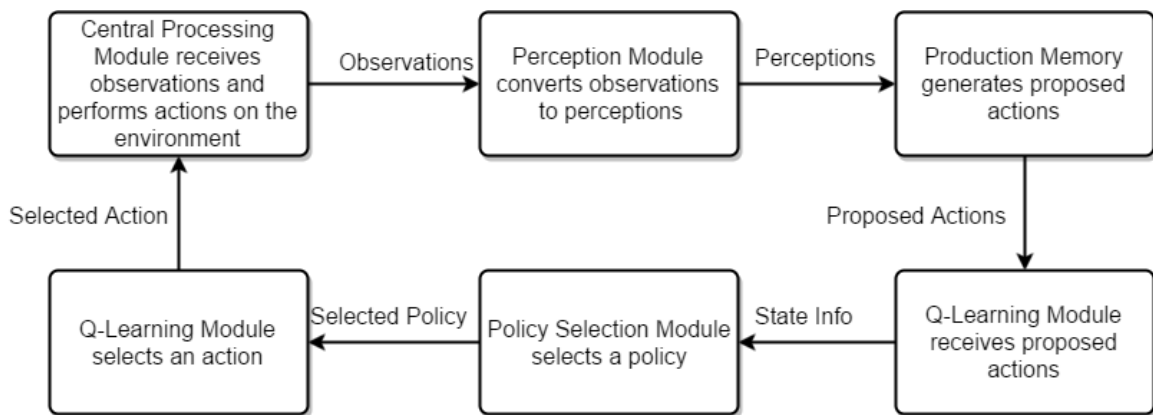


Figure 3-2: A diagram outlining the decision cycle of Q-Cog.

3.3 Policy Selection

A policy is a rule that an agent may follow in order to select an action given the current environment state. Policies map environmental states to actions that agents

may perform [42]. The architecture maintains a library of policies and allows for the selection of a single policy to execute and refine at a given time. In this proposed design, each policy consists of a single Q-Table that may be modified when reinforcement values are received from interactions in the environment. The pseudo code for the Q-Learning Module with the addition of policy selection is shown in Listing 3.1. Individual Q-Tables are stored within the Policy Selection Module and are refined and utilized by the Q-Learning Module.

Listing 3.1: Q-Learning Module and Policy Selection Module Interaction

```
Update Current State Information
Receive Proposed Actions
ReadEpisodicMemoryTransitions()
BestAction = PolicySelectionModule.SelectAction()
PolicySelectionModule.UpdateQValues()
UpdateEpisodicMemory()
return BestAction
```

The number of policies in Lample and Chaplot's [24] system did not change over time. The mechanism within Q-Cog allows the agent to maintain many specialized policies to handle different areas of the environment as opposed to maintaining a single general policy. The Policy Selection Module extends the Q-Learning Module by storing and selecting learned policies during execution. At each time-step the most applicable policy, as defined by either a learned or pre-defined mapping of situations to policies, is selected for execution. This does not force the agent to finish learning a complete policy before learning another. The agent is able to maintain multiple policies and refine them independently. A rule base may be used to define the conditions for policy selection. When the conditions of a rule are met, the policy specified by the output of the rule may then be selected. Rules can be pre-inserted by developers or a rule compilations system may be designed and integrated for automation of the process.

As depicted in Figure 3-3, the module is able to store all learned policies in memory and maintain a reference to the current active policy for action selection and learning. The Q-Learning Module may then work with the current active policy during learning and action selection. When reinforcement values are received from the environment, these values are applied to the current active policy by the Q-Learning Module.

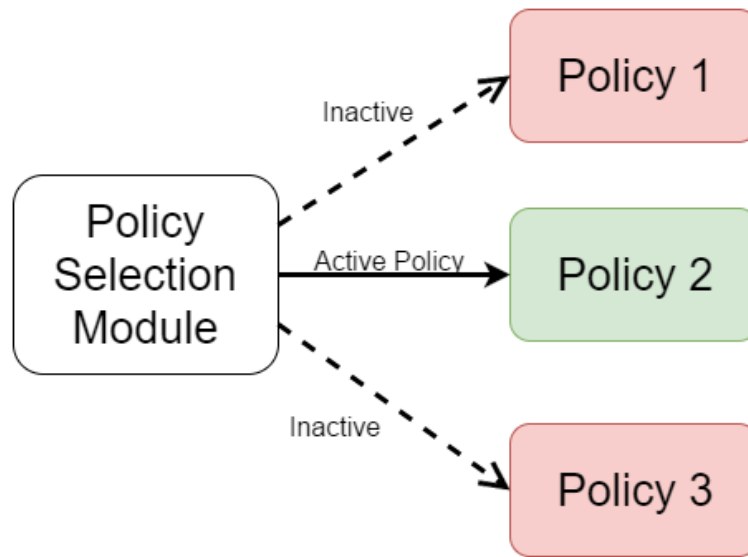


Figure 3-3: The situation-policy mapping within the Policy Selection Module.

The diagram in Figure 3-4 depicts the work flow of the Policy Selection Module. The flow is as follows:

1. State information is received from the Q-Learning Module and used to select an appropriate policy. This selection is based on the conditions of a rule set. When the condition of a rule matches state information, the policy specified by that rule is selected for use by the Q-Learning Module.
2. If no applicable policy is found during this process, a new policy is generated and stored within the policy library.
3. Either the selected policy from memory or the newly generated policy is set as the current active policy and will be updated by Reinforcement Learning using Equation 3.1 and used during action selection.

- Utilizing the current active policy, the optimal action is selected based on the current state of the environment.

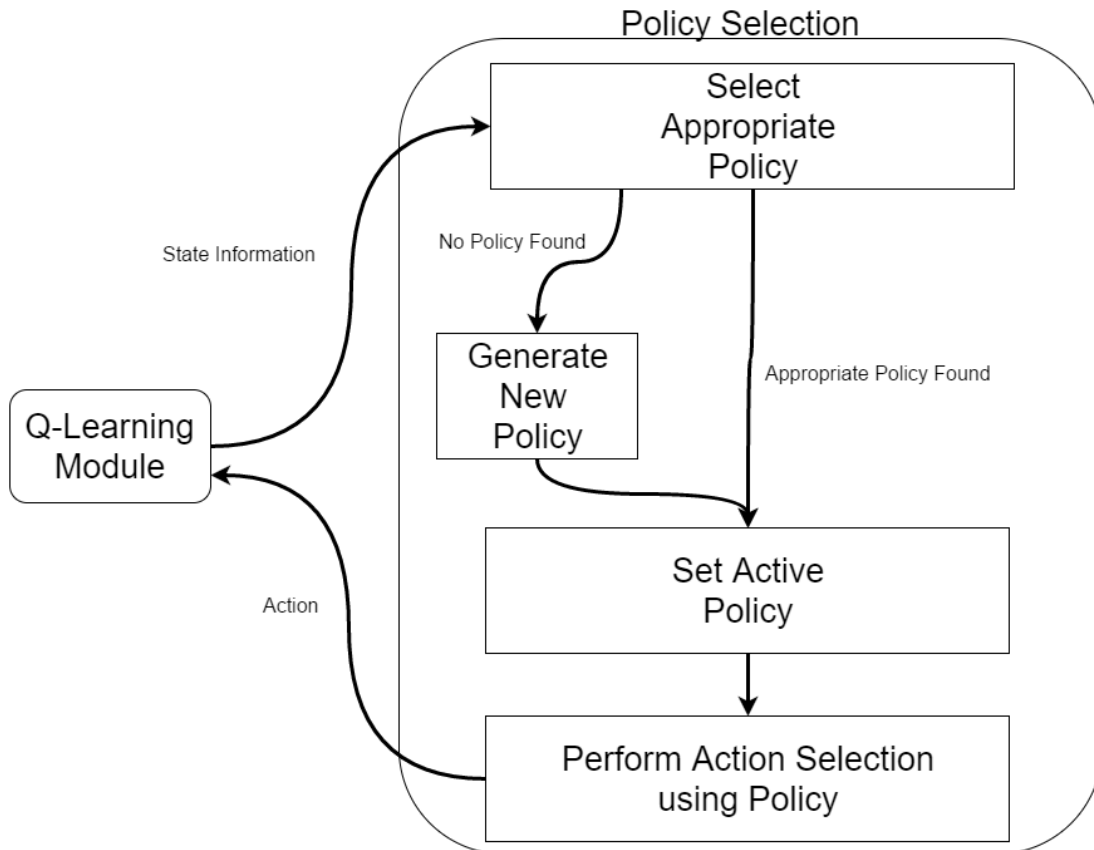


Figure 3-4: The Policy Selection Module.

3.4 Reference Q-Cog Implementation

The following section outlines the reference implementation of the Q-Cog architecture. The architecture was designed, implemented and tested through an iterative process. Testing was performed in a developed experimental platform (Chapter 4) using various crafted scenarios. These scenarios aimed to measure how effectively the architecture adapted to environment changes.

The architecture was implemented using the Java programming language. The following sections outline the implementation of each of the modules of the architecture. Examples used in this section draw from the proposed scenario involving a

humanoid agent acting within a 3D virtual environment containing hostile predator agents. This scenario is elaborated further in later chapters.

3.4.1 Module Implementation

An overview UML diagram of the architecture is depicted in Figure 3-5.

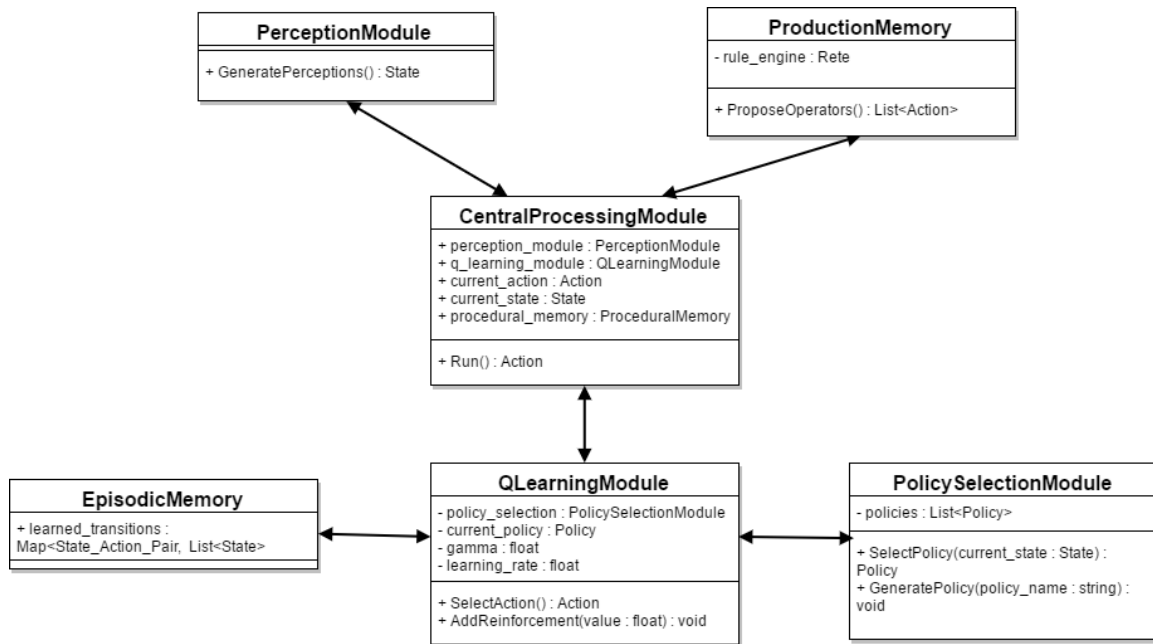


Figure 3-5: A UML diagram of the reference Q-Cog implementation.

The Central Processing Module

The Central Processing Module controls the flow of information between other modules. It therefore contains a reference to each of these modules in order to obtain the output of each module and to supply the input to the next module in the decision cycle. The *Run()* method iterates through the decision cycle that involves generating perceptions, proposing actions from these perceptions and finally selecting an action to perform utilizing a policy. For example, it would invoke the *GeneratePerceptions()* method within the Perception Module when in the second step of the decision cycle. The *Run()* method returns an action obtained from the Q-Learning Module as a result of the action selection process. This action is then performed on the environment

and the decision cycle is run again from the beginning. Additional environment data, such as reinforcement values, are firstly received by the Central Processing Module and then routed to their respective destination modules.

The Perception Module

The function of this module is to translate environment observations from the Central Processing Module into a set of perceptions that collectively make up the current state. Although these observations are case specific, in most cases within the virtual 3D domain they are 3D vectors encoding directions or agent world locations. These vectors are of the form (x, y, z) .

A state is generated in the *GeneratePerceptions()* method and returned to the Central Processing Module to be passed on throughout execution. The current implementation of the observation discretization process involves a set of predefined rules governing the translations. Example state information includes danger and health indication values of a humanoid agent. Given environment observations relating to hostile agents in the environment, the Perception Module will generate a discretized danger value based on aspects such as the collective distance to each of these hostile agents. A new state will then be returned by the module containing this danger value along with other generated state information. A simplified example of such a rule pertaining to humanoid health is as follows:

Algorithm 1 A Depiction of the Discretized Health Value

```
if Health <= 20 then  
    CurrentState.HealthIndicator = 0  
else if Health > 20 & Health <= 60 then  
    CurrentState.HealthIndicator = 1  
else  
    CurrentState.HealthIndicator = 2  
end if
```

Future implementations may utilize various techniques, such as Artificial Neural Networks, to automate the discretization process of observations.

The Production Memory Module

Production rules within the Production Memory Module are represented using the Jess Rule Engine in the Java programming language. The Jess Rule Engine provides a rule-based language to define various production rules. These rules are sets of condition-action statements that utilize certain facts to govern when the action of each rule is performed. The rule engine may then be extended with further production rules without requiring changes to application logic.

The Production Memory Module contains a reference to a rule engine that contains production rules relating to agent actions. Using the state information supplied as input by the Central Processing Module, a list of proposed actions may be inferred from the rule engine within the *ProposeOperators()* method. The example Jess rule below proposes the *Engage Predator* action if a predator has been sighted and the predator is still alive. This does not necessarily mean that the action will be selected for execution, the action is merely proposed to be considered in the decision making process.

Listing 3.2: An Example Jess Rule

```
(defrule PredatorSighted
"A rule defining how to engage a sighted predator"
?o <- (Predator {health > 0})
=>
(bind ?action_list (new ArrayList))
(?action_list add (new Symbol 9)) ;; Action 9 = Engage Predator
(add ?action_list))
```

The production rule above may be translated into *if-then* statements as follows:

```
List<Action> ProposedActions
Observation 0 ; Inserted Observation

if (0.Predator && 0.Predator.Health > 0) then
    ProposedActions.Add(Action.EngagePredator(0.Predator))
```


For the purpose of this implementation, Jess production rules were pre-developed and were not modified at runtime by the agent. Future implementations of the architecture may include production compilation techniques similar to that in the ACT-R architecture.

The Q-Learning Module

The Q-Learning Module utilizes policies in order to select an action. Each policy is represented as a table with each table containing the Q-Values for each state along with each action that may be performed in that state. The example section of a policy shown in Table 3.1 pertains to a humanoid agent needing to select an action when state information indicates a medium environment danger level, a low health indication and there is an available food source nearby. The first three columns represent state information, the fourth column indicates the action identifier and the last column is the corresponding Q-Value. In this particular example, the highest Q-Value correlates to the *Retrieve Food* action in row three of the table and so this action will be selected for execution. Based on the outcome of this action, reinforcement values received will then be used to refine this table entry within this policy.

Table 3.1: A table depicting an example section of a generated policy.

Danger Indicator	Health Indicator	Food Indicator	Action	Q-Value
Medium Danger	Low Health	Food Available	Explore	0
Medium Danger	Low Health	Food Available	Attack	-4.183
Medium Danger	Low Health	Food Available	Retrieve Food	3.1
Medium Danger	Low Health	Food Available	Flee	-5.465

The *SelectAction()* method is the main control method in the Q-Learning Module. The execution order within the *SelectAction()* method is as follows:

1. Select a policy using the Policy Selection Module reference and state information. If no policy exists given the supplied state information, generate a new policy and set it as the current active policy.

2. Using the current active policy and proposed actions from Production Memory, select an action based on Q-Table values. The proposed action with the highest Q-Value with regards to the current active policy is selected for execution.
3. Return the selected action to the Central Processing Module to be performed on the environment.

The Episodic Memory Module

Episodic Memory stores all state transitions in a hash map that maps state-action pairs to resulting states i.e. $F : s, a \rightarrow s'$. Given a state action pair $\langle s, a \rangle$, the function returns a list of possible resultant states. This then allows the Q-Learning Module to infer the list of possible resultant states that the agent may move to given that a particular action is performed in a particular state. A simple example of a state transition may include a humanoid agent moving from a state of high environment danger to a low danger state when the *Flee* action is performed. Whenever the state of the agent changes, a check is done to verify if the current state-action pair contains a mapping to this new state. If no mapping exists, a mapping to the new state is stored within this module.

3.4.2 The Policy Selection Module

The first requirement of the Policy Selection Module is to store all policies available to the agent. These policies are stored in a list which may be enlarged and reduced with ease at runtime. These policies are then accessed when a request is received by the Q-Learning Module to select a policy. When this request is received, the Policy Selection Module extracts relevant state information in order to identify a policy to select. A hash map that maps state information to policies will retrieve and return the relevant policy. If no policy exists, a new policy is generated and the corresponding state information is used as the key of the hash map which will link to this new policy.

When a new policy is generated, all Q-Values in the table are defaulted to a value

of 0. The output is therefore a blank policy that will be refined once activated. When the agent activates this policy for the first time it continues the exploratory process of performing actions and generating Q-Values for the new policy.

3.5 Summary

The proposed architecture aims to provide a mechanism to develop adaptive, self-learning agents that may utilize dynamic policy generation and selection within the 3D virtual world domain. Various concepts adopted from the state-of-the-art cognitive architectures SOAR, ACT-R and CLARION were integrated with Q-Learning and a policy generation mechanisms to fulfill the listed architecture requirements.

Chapter 4

Design and Implementation of the Experimental Platform

The following chapter outlines the implementation details of the experimental platform used to evaluate Q-Cog agents. The platform is reusable and provides support for developing and experimenting with agent-based simulations in 3D virtual worlds. The chapter firstly outlines the requirements of the platform. Secondly, the support provided by the platform is discussed. Thirdly, the mechanism used to link agent architectures to the platform is described. Lastly, an overview on how to use the platform is provided and scenario-specific extensions are described.

4.1 Platform Requirements

The aim of the platform is to provide the necessary support for developing and evaluating agent-based models. The required components and specifications of the platform are as follows:

1. Provide the necessary support for engineering and building agents.
 - (a) Sensory support.
 - (b) Actuator support.

2. Provide the required tools to allow for the development of the 3D environment.
3. Provide simulation support allowing users to run specialized experiments utilizing custom agent architectures.
4. Contain generic, extensible components that may be customized to suit the requirements of the user.

4.2 The 3D Experimental Platform

The following section outlines the details of the reusable 3D experimental platform designed to provide an environment for agent development with sensors and actuators. The experimental platform was implemented using the Unity3D Game Engine [43]. The Unity Engine was selected as it provided tools for 3D rendering, agent pathfinding algorithms and a physics engine. These Unity3D tools were utilized to develop the specialized platform based on the listed requirements. Figure 4-1 depicts an example simulation within the experimental platform.

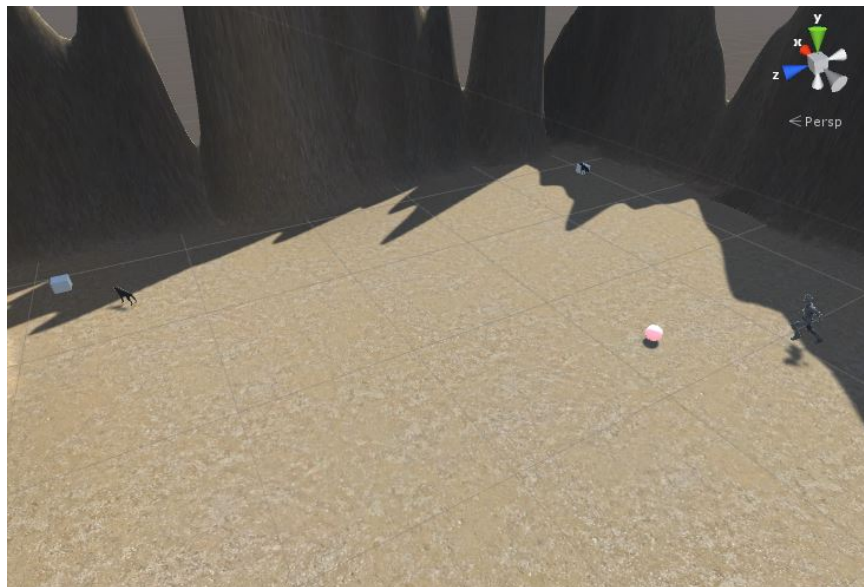


Figure 4-1: An example simulation within the experimental platform.

4.2.1 The Generic Platform

The experimental platform provides various components for the development of agents including sensory support and actuators. The following subsections provide details on each aspect of the platform.

Agent Development Support

The experimental platform provides the following support for agents:

- Customizable sensory support allowing agents to observe the environment.
- Navigation support allowing agents to traverse the environment.
- Item collection and storage allowing agents to collect and utilize specified world entities.

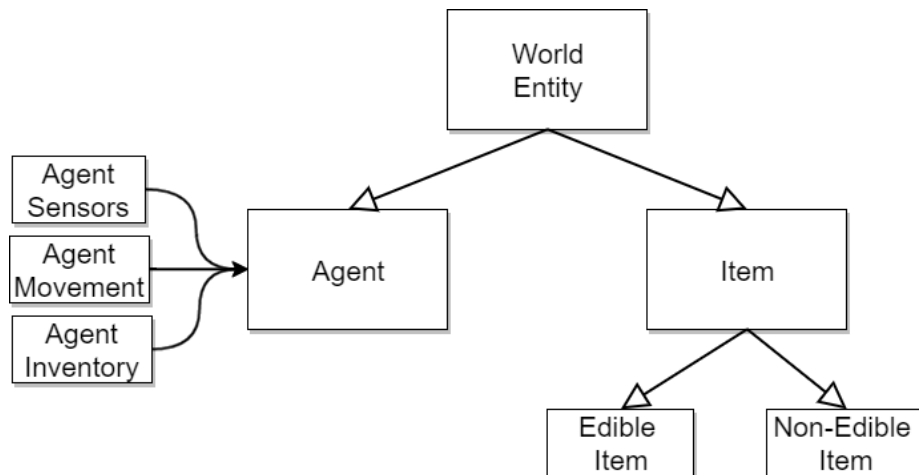


Figure 4-2: A diagram providing an overview of entities within the experimental platform.

A *world entity* within the experimental platform defines any object placed in the environment. The diagram in Figure 4-2 provides an overview of the entity system within the platform. Each agent maintains certain components allowing it to act within the environment. Other world entities, such as items, contain information relating to how agents interact with them. For example, an agent may collect an

edible item and store it within the inventory component. This item may then be utilized by the agent at a later stage.

The entity system and entity components have been designed to be generic and extensible. Scenario-specific entities are therefore expected to be defined to suit experimental requirements.

Sensory Support

Agent sensors were designed and implemented in order to observe world entities in the environment within a certain range and only when in line-of-sight. This allows objects in the environment to block the vision of agents and force additional exploration. Complete environment information is not available to agents at the start of each simulation. Deformations in the environment may also block agent vision and hence provide higher complexity for the agent.

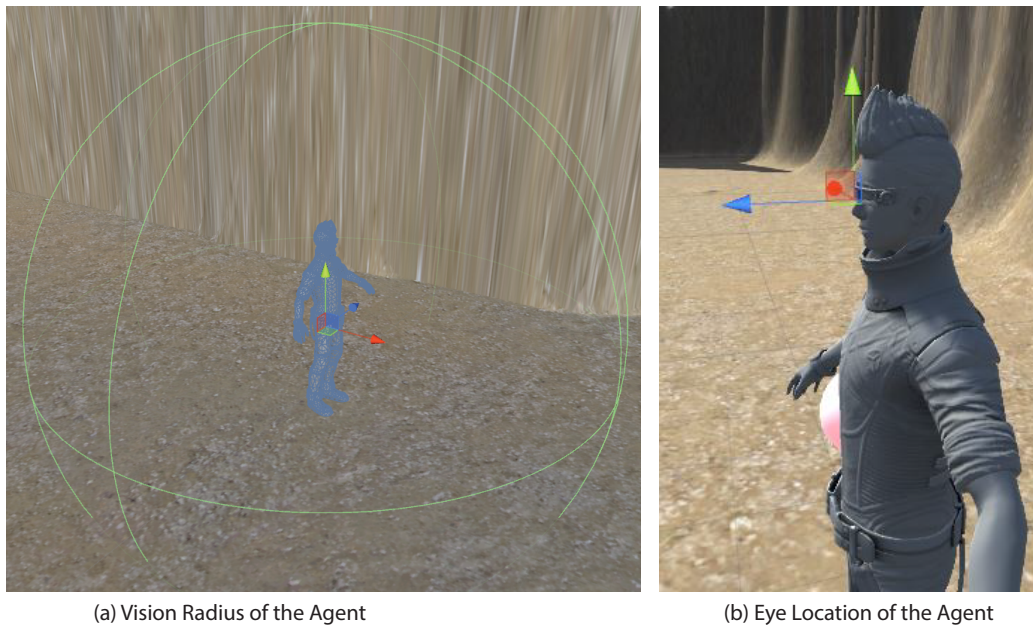


Figure 4-3: The agent's vision.

The vision radius of the agent may be controlled by editing the radius of a sphere in a manner similar to that seen in Figure 4-3. This radius may be increased or decreased as required for each experiment or simulation. Figure 4-3 also provides an example depicting the eye location of an agent in the experimental platform. This

eye location, along with the vision radius, is used to calculate whether an object is currently visible via a technique known as raycasting. At each update, a number of rays are projected from the eye location to each potentially visible entity in the world. The projection distance of each ray is indicated by the sight radius of the agent and the origin of the ray is defined by the agent's eye location. The percentage of rays that hit the target entity indicate how certain the agent is that the entity is actually at that location. An outline of the sensory process of environment agents is as follows:

1. Once another entity has entered the vision radius of the agent, that entity is marked to be tested for line-of-sight visibility.
2. A number of rays are then fired from the agent's eye location to each of the marked entities. The number of rays projected determines the accuracy of the sensor and is modifiable before each simulation.
3. The percentage of rays that successfully intersect the target entity without first intersecting other entities indicates how certain the agent is that the entity is actually present at that location (i.e. the confidence rating). A confidence rating of zero immediately marks the entity as not visible.
4. Each entity, along with their corresponding confidence rating, collectively make up the current visible entities for the agent.

Actuator Support

Actuator support was provided in the form of a navigation system. Agents traverse the world by moving to navigable areas defined by a navigation mesh. This navigation mesh is generated by the Unity3D engine. The supplied Unity3D pathfinding system then allows agents to plan an optimal path to reach a specified point in the 3D world. A depiction of the navigation mesh in the environment is shown in Figure 4-4. In this image, supplied by the Unity3D engine developers, the blue area highlights the generated navigable regions of the world. The cylindrical agent is then able to move to any location within this region via pathfinding algorithms.

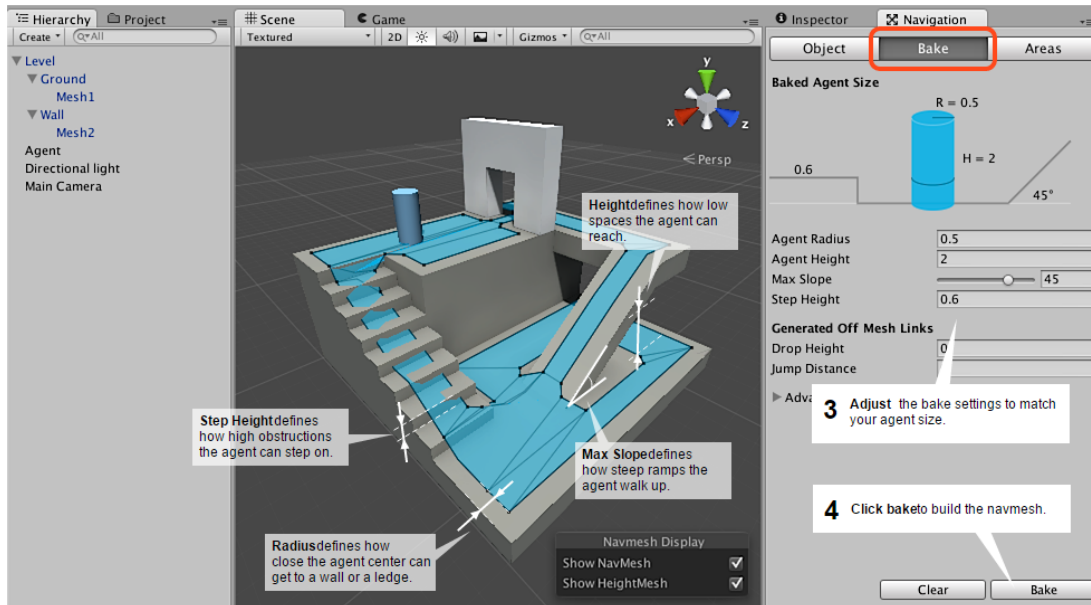


Figure 4-4: A depiction of the navigation mesh in Unity3D. From: docs.unity3d.com/Manual/nav-BuildingNavMesh.html

The Architecture Connector

The experimental platform utilizes a connector in order to communicate with a connected agent architecture. This satisfies the requirement of a plug and play feature as new architectures may utilize the platform through the connector with minimal configuration. This was achieved through a Transmission Control Protocol (TCP) socket connection as is shown in Figure 4-5 and the broad platform overview is shown in Figure 4-6. The connector receives incoming observations from the experimental platform while transmitting selected actions, selected by the concrete agent architecture, to the platform for execution. An advantage of this approach is the possibility of running the experimental platform and the agent architecture implementation on separate physical devices in order to improve performance.



Figure 4-5: A depiction of the TCP connection from the experimental platform to the agent architecture.

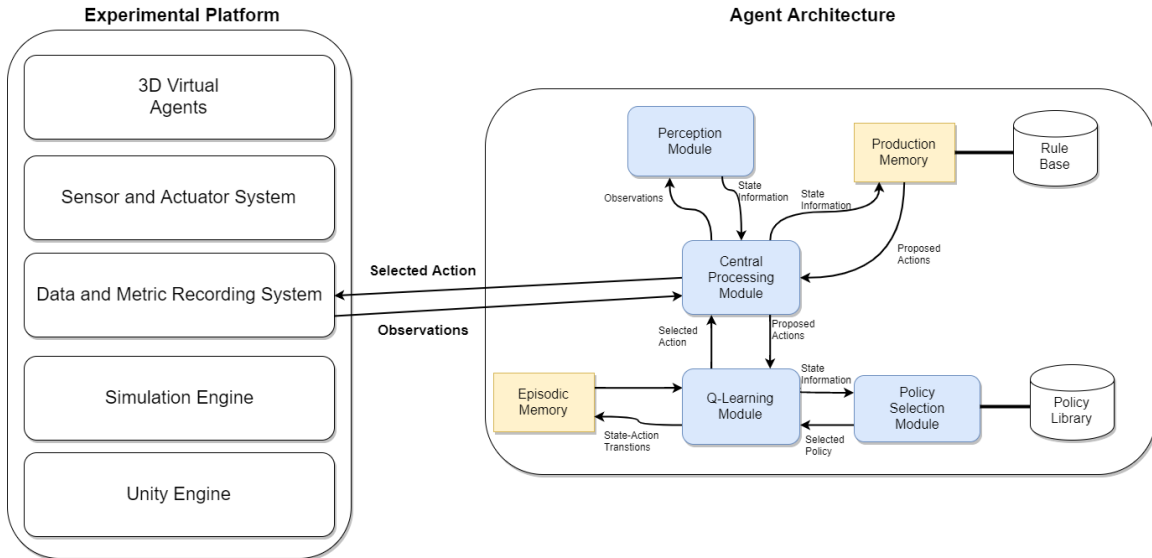


Figure 4-6: An overview of the experimental platform architecture.

Once information has been received via the TCP connection, a complete model of the environment is constructed based on each agent’s observations. The specific architecture in use may then retrieve this information and utilize it during action selection and learning. Once a specific action has been selected, the architecture notifies the connector to transmit the action details to the experimental platform to be carried out by the agent.

Transmitted Data

All agent observations are transmitted to the agent architecture from the experimental platform to be used during reasoning. These are raw observations containing visible entity locations in the world and agent attribute information. Consider the example situation depicted in Figure 4-7. Agent *A* and an item *F* are visible to the humanoid agent. Agent *B* however is blocked due to deformities in the terrain. Location information of the item and agent *A* will therefore be the observation data transmitted to the agent architecture and information relating to agent *B* is not transmitted.



Figure 4-7: An example of observation data transmission.

4.2.2 The Simulation Engine

The platform is required to provide support for various simulations to be performed. This includes features such as: Running an experiment for a certain number of iterations, recording important metrics and replaying simulations after completion for further analysis. This support is controlled by the simulation engine section of the platform as depicted in Figure 4-8.

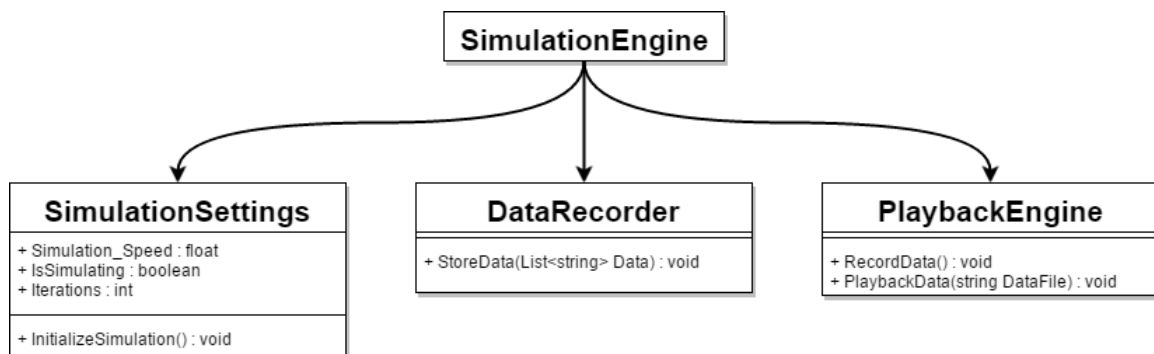


Figure 4-8: An overview of the simulation engine in the experimental platform.

Simulation Settings

A simulation is parameterized by various settings. Firstly, the speed of the simulation must be defined. This determines the rate at which world entities traverse the world

and perform actions. This allows for a lengthy simulation to be completed at a faster rate. The length of the simulation is defined by the number of iterations performed. Once an iteration is completed, world entities are reset to their initial states and a new iteration is started.

Metric Recording

The data recorder allows various metrics, such as success rate, to be recorded during each simulation. Users may then define the points in the execution for which they require data to be stored. Once the simulation has been completed, the data file containing this information is output to the user for analysis.

Recording and Playback

The playback engine allows simulations to be recorded and played back at a later stage for further analysis. The engine constantly records the attributes and actions of every world entity during a simulation. Once a simulation has completed, the recording is stored. Users may then load a particular recording file to playback the simulation. The data pertaining to each world entity is stored in a separate file. Each file is then loaded by the playback engine when performing a playback.

4.3 Platform Customization and Setup

The following section provides an overview of how the generic experimental platform may be utilized.

4.3.1 Agent Development

In order to develop a custom agent within the experimental platform, the user must firstly provide the appropriate 3D model of the agent within the Unity3D editor as depicted in Figure 4-9. This is only a visual customization and does not effect agent behaviour.



Figure 4-9: A 3D humanoid model in the experimental platform.

Secondly, the desired components of the agent should be defined as per Unity3D object creation protocol. The defined components of a humanoid agent are shown in Figure 4-10. The component outline is as follows:

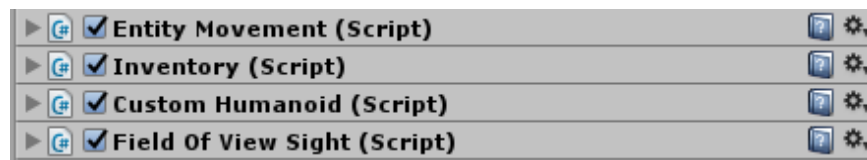


Figure 4-10: The components of a humanoid agent within the experimental platform.

1. The Field of View Sight component is the provided platform sensory support. This defines how the agent is able to observe the environment by defining attributes such as vision radius and the eye location.
2. The Entity Movement component specifies movement information of the agent. Aspects, such as movement speed, relating to how the agent traverses navigable terrain are defined here. An example code snippet is shown in Figure 4-11.

3. The Inventory component monitors the item collection aspect of the agent. This includes the maximum carrying weight of the agent and references to the currently collected world entities.
4. The Custom Humanoid component contains the specialized agent behaviour that defines any unique aspects relating to how the agent interacts with other world entities.

```
/* Move to a location with a specific speed percentage ...  
public void MoveToLocation(Vector3 new_location, float speed_percentage = 1)  
{  
    current_move_location = new_location;  
    current_speed = original_speed * speed_percentage;  
    current_speed_percentage = speed_percentage;  
    nav.SetDestination(current_move_location);  
}
```

Figure 4-11: An example code snippet from the Entity Movement script.

4.3.2 Setting Up a Simulation

The following are the steps required to setup a simulation within the experimental platform:

1. Develop the required environment setup within the Unity3D editor by using a number of 3D models.
2. Using the navigation mesh tool as depicted in Figure 4-4, define the navigable areas of the developed terrain. This completes the development of the 3D environment.
3. Develop a number of agents with specialized behaviour as is outlined in section 4.3.1 and place agents in appropriate locations in the world.
4. Define the simulation settings such as iteration count and simulation speed.

5. Launch the implemented agent architecture and then launch the experimental platform simulation.
6. Once an iteration has completed, agents will be reverted back to starting locations and metrics will be recorded.

4.4 Summary

The developed experimental platform contributes a reusable and extensible means of experimenting with agent-based simulations in the 3D virtual world domain. The platform provides the necessary support such as sensory and navigation tools. These are combined with a simulation structure containing metric storage, simulation recording and playback support to supply a full experimental platform package.

Chapter 5

Experimental Design and Results

The usability of the experimental platform and the adaptability of Q-Cog agents were evaluated. This chapter describes the empirical evaluation of the adaptability of Q-Cog agents and evidence for the usability of the experimental platform is also provided. The platform was successfully used by other research projects involving many different areas of artificial intelligence: Ontologies, Bayesian Networks, Deliberative and Reactive Architectures and Deep Learning [2, 16]. This variety aimed to show that the platform may be used in many different areas of AI research. The hypothesis tested for Q-Cog was that a policy library will improve the adaptability of Q-Cog agents. The platform was used to setup scenarios within which the agents were expected to survive against hostile agents. Q-Cog agents with and without the policy library were evaluated and it was found that the agent utilizing the policy library had a significantly higher performance as the complexity of the environment was increased.

5.1 Scenario Overview

The hypothesis tested was that Q-Cog agents utilizing a policy library will adapt to changes in the environment better than agents without the policy library. Several experiments were carried out by pitting agents with and without the policy library against hostile agents with varying combat strengths. The scenario is that a humanoid

agent must adapt to changing circumstances to survive. The scenario consisted of a number of hostile predators that need to be eliminated through combat simulations while available food sources provide a source of regeneration during or after combat. The humanoid agent is required to adapt its behaviour and gather knowledge about the environment in order to overcome certain challenges discussed below. The predator is a hostile agent that wanders the environment and whose primary goal is to eliminate the humanoid when sighted. Agents were required to reason about threats and food sources and make optimal decisions when to fight or flee and when to consume food. The scenario was setup within the complex 3D virtual world experimental platform (Chapter 4).

When a predator sights the humanoid in the world, it attempts to engage it in combat. The environment contains predators of varying strengths. The strength of the predator is determined by health and combat damage attributes which indicate their combat effectiveness. An objective of the humanoid agent is to distinguish between different predators and adapt its behaviour appropriately to effectively eliminate them in combat.

The goal of the humanoid is to eliminate all predators in the world while avoiding death. The humanoid fails a simulation when it is eliminated in combat. All agents in the world contain a health attribute. The act of combat decreases health and when the health value of an agent reaches zero, the agent is eliminated. Consuming food is the only method of regaining health. The humanoid is required to consume food which is present around the world in order to regain the health lost as a result of combat. This food is available only for the humanoid agent. The quantity of food is finite and so the humanoid must carefully choose when it is most appropriate to consume the available food. The negative implications of consuming food at the incorrect moment results in potential health being lost as the humanoid may not exceed its starting total health value. Therefore the optimal food collection process involves only consuming food after combat. With regards to predator combat, the appropriate actions include disengaging from combat and seeking food when dealing with stronger predators as they may not be eliminated in a single combat scenario.

5.2 Experimental Platform Extension: Agent Combat

To accommodate the requirements of the scenario, the experimental platform was extended to include a combat sub-system allowing agents to engage each other in realtime combat. Firstly, agents were modified to allow for an assigned health value. This value determines how close the agent is to dying. Once the health value of an agent reaches zero, the agent is eliminated until the next run. Secondly, a combat component was added to the platform defining how agents may engage each other. This combat component maintains a particular damage attribute as well as an attack frequency attribute. The damage attribute indicates how much health is removed from the other agent as a result of an engagement with this agent during combat. The attack frequency attribute determines the interval at which the damage attribute is applied to the health of the opposing agent. A typical engagement involves:

1. Agents first move within hand-to-hand range of one another.
2. In realtime, each agent removes a certain amount of health from the other agent based on their combat damage attribute.
3. If the corresponding health indicator of either of the agents reaches a value less than or equal to zero, that agent is eliminated and removed from the current run.

Agents are able to disengage from combat at any time and continue to perform other tasks in the world. This does not prevent the other agent from continuing combat. For example, if one agent decides to flee during combat, the second agent may pursue and continue to attack. Figure 5-1 shows two agents engaged in combat.

Predator agent behaviour was defined by a Finite State Machine. Predators engage the humanoid when within a certain radius. When the humanoid moves out of this radius, the predator will disengage. The disengage radius is randomly generated for each predator which provides a degree of non-determinism.

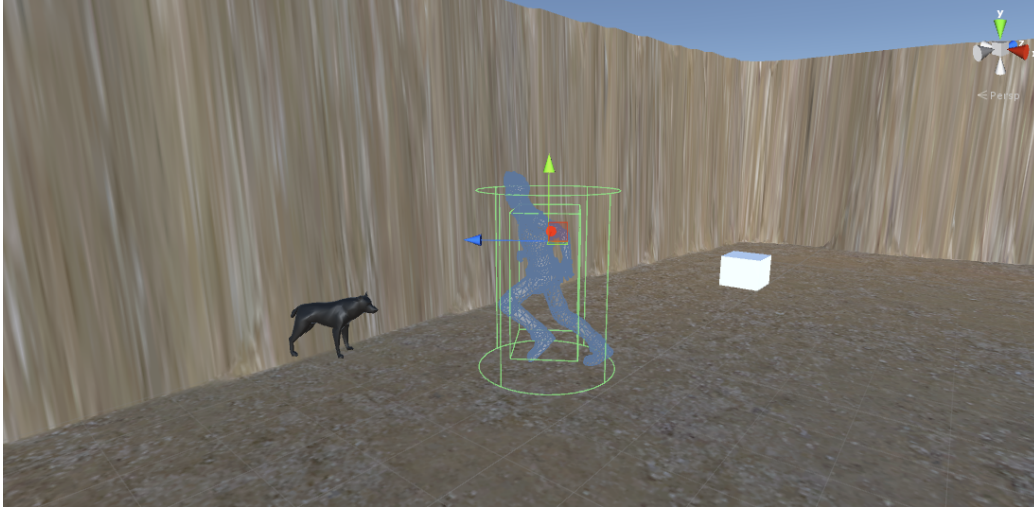


Figure 5-1: An screenshot from a simulation in the experimental platform where two agents have engaged in combat.

5.3 Experimental Design

5.3.1 Environment Design

Environment Rewards

The Q-Cog architecture utilizes Reinforcement Learning and so reinforcement values were associated with important environment events. Whenever the humanoid agent performs an action on the environment, an immediate reward value is given as feedback. The reward values for each environment event are available in Table 5.1. These reward values were finalized prior to experimentation setup. A set of initial values were selected as the starting point and a number of tests were run to calibrate the simulations.

Table 5.1: Reward values received as a result of each environment event.

Environment Event	Reward Value
Predator Eliminated	+2
Humanoid Killed	-4
Food Consumed at Appropriate Time	+2
Food Consumed and Wasted	-2

The reward values with regards to food consumption in Table 5.1 are calculated based on whether or not food has been wasted. Food is wasted when the agent consumes an item of food when it is at maximum health and does not require it.

Environment States

Various properties in the environment, such as distances to predators and location of food are continuous variables. Thus, the environment, at any time could be in an infinite number of states. To make the learning problem tractable, the state space was discretized. These states are shown in table 5.2. The danger level of *Low* indicates that there are no nearby predators while a danger level of *High* indicates one or more predators nearby. Humanoid health represents the three numeric health values that the humanoid may have (*Low* = 1, *Medium* = 2, *High* = 3). The food available may either be *True* or *False* and indicates if food is in the immediate vicinity of the humanoid.

Table 5.2: A table representing the discretized state space.

State Number	Danger Level	Humanoid Health	Food Available
1	Low	Low	False
2	Low	Low	True
3	Low	Medium	False
4	Low	Medium	True
5	Low	High	False
6	Low	High	True
7	High	Low	False
8	High	Low	True
9	High	Medium	False
10	High	Medium	True
11	High	High	False
12	High	High	True

In each experiment, the agent refines an optimal activity selection policy to ensure that it survives for the maximum amount of time that it can and defeats all predators in the environment. A policy is an optimal mapping from each state, listed in Table 5.2, to one of the following activities:

1. The **Explore** activity involves the agent searching random areas of the envi-

ronment in order to find either food or predators.

2. The **Engage** activity involves the agent engaging the closest visible predator in combat.
3. The **Flee** activity involves the agent running away from surrounding predators.
4. The **Eat** activity involves the agent approaching and consuming a nearby visible food source.

5.3.2 Metrics

Two metrics were used to measure the performance of the agent: success rate and survival time. The success rate indicates the percentage of predators eliminated by the humanoid during each run. The calculation of the success rate is shown in Equation 5.1.

$$\text{Success Rate} = \frac{\text{Eliminated Predator Count}}{\text{Initial Predator Count}} \quad (5.1)$$

An example dataset containing success rates is shown in Figure 5-2. This contains a portion of data indicating the success rate of the humanoid after each run. Each value indicates the percentage of predators eliminated during the run with 100% indicating that all predators in the environment were eliminated. The average success rate of the humanoid during this simulation may be plotted from this data.

The survival time indicates the length of time per run that the humanoid is able to survive in the environment without being eliminated. At the start of each run, a timer is started which records this metric. When the run is completed the time is recorded. A run is completed when either the humanoid is eliminated or all predators are eliminated.

5.3.3 Setup and Execution of Simulation

The experimental platform allows for easy integration and modification of different scenarios. The Prefab system within the Unity3D engine allows various object types

Iteration	Success (%)
1	0
2	0
3	50
4	100
5	100
6	100
7	100
8	100
9	100
10	100
11	100
12	100
13	100

Figure 5-2: A figure depicting a portion of a dataset containing simulation results.

to be saved for later use. This allows researchers to quickly switch out agents in the Unity editor and begin new experiments. An example of a scenario setup is shown in Figure 5-3. In this depiction, three predator prefabs and the humanoid prefab are available to be inserted into the environment. Any number of predators may be placed in the world at different locations.

This experimentation begins each run by randomly placing predators and food sources in the environment subject to a minimum distance constraint of 30 Unity units. Predators are randomly placed in the environment with a minimum distance between other predators. A similar generation system is used for food sources. This distance constraint was implemented to prevent the generation of clusters of predators or food sources in the environment. Once the environment is generated, the humanoid is placed at a designated starting location and the experimentation is started.

5.4 Experiments and Results

Experiments were carried out to demonstrate that:

1. The Q-Cog architecture allows the agent to learn an optimal survival strategy in a complex 3D virtual world.
2. The policy selection mechanism allows the agent to adapt effectively to dynamic

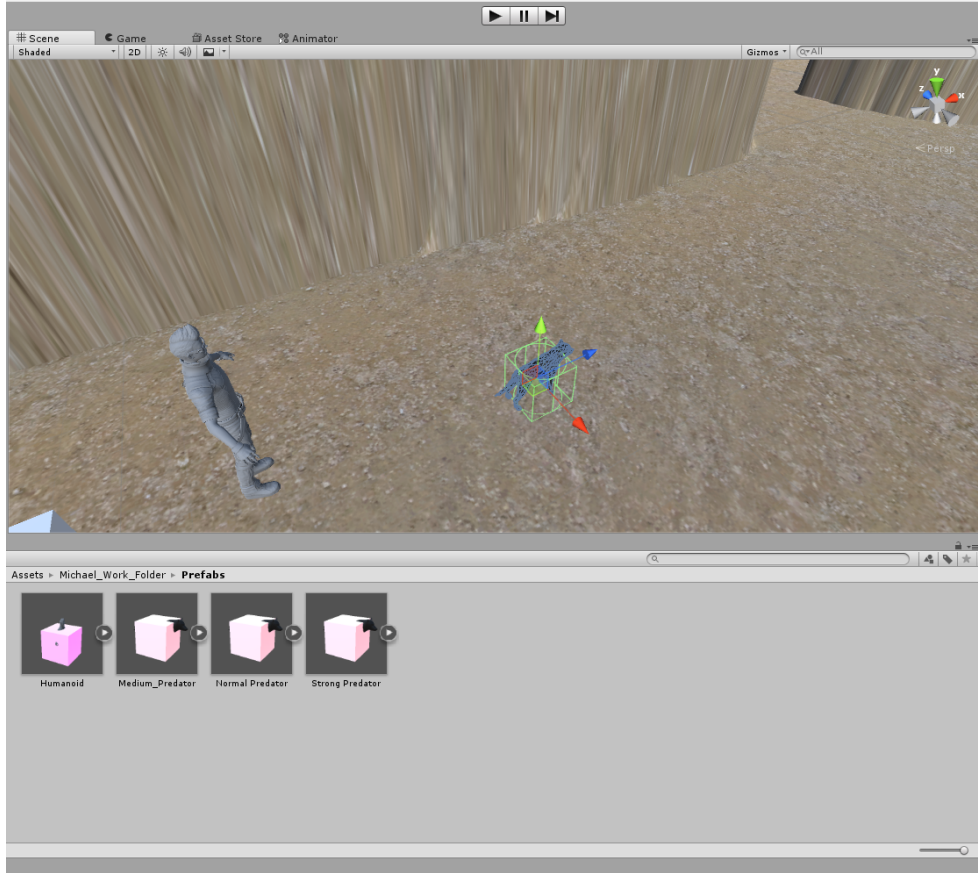


Figure 5-3: Prefabs in the Unity engine allows various object types to be stored for later use and added to the scenario.

environments.

3. The experimental platform may be used successfully to test and evaluate cognitive agents in virtual 3D worlds.

The reference Q-Cog implementation is used to control the behaviour of the humanoid agent within the world given various experimental scenarios defined. In all experiments performed, the humanoid agent must eliminate a number of predators while consuming available food sources at the appropriate time in order to stay alive. The humanoid agent contains no initial knowledge of the environment. The agent is required to learn the results of its own actions on the environment and is unaware of the nature of other agents. The knowledge acquired by the agent is re-utilized in each run and so the agent is able to refine a policy during an experiment.

5.4.1 Experimentation

Six experiments were conducted with each experiment consisting of sixty runs. This allowed for three parameter sets to be run twice over. The experimental parameters are as follows:

- The **Run Count** defines the number of complete runs of each simulation. A run is completed when the humanoid agent is either eliminated or has eliminated all predators in the environment.
- The **Predator Count** defines the number of predator agent in the environment at the start of each run.
- The **Predator Type Count** parameter defines the variety in predator behaviour. For example, a type count of three indicates that the initial predator count will be populated using three different predator types. The number of predators from each type will always be equal. Each predator differs in health values and combat damage values.
- The **Available Food Sources** parameter defines the number of food sources distributed around the environment at the start of each run.

Experiment Objectives

This experimentation process aims to provide a performance comparison between agents utilizing policy selection versus agents without policy selection.

Experimental Parameters

Three experiments were conducted with the parameter sets as shown in Table 5.3. Each experiment was performed with and without policy selection, resulting in a total of six experiments. The run count, number of predators and food sources were kept constant and only the number of predator types was varied to test the adaptability of agents. The three predator types used, namely types A, B and C, vary in strength with predator type A being the weakest and C being the strongest.

Table 5.3: Experimental Parameter Sets.

Parameter Name	Set 1	Set 2	Set 3
Run Count	60	60	60
Predator Count	6	6	6
Available Food Sources	3	3	3
Predator Type Count	1	2	3

Increasing the predator type count parameter increases the complexity of the environment for the humanoid agent. A consistent predator count of six was selected to allow for an even distribution in the amount of predators of each type as the predator type count attribute was increased. For example, when the predator type count attribute is three, there will be two predators of each type in the environment at the start of each run. The food source parameter was kept at half the predator count to force the humanoid to correctly preserve the supply.

Results

The policy library and policy selection extension to the architecture allowed the agent to develop separate strategies for each predator as illustrated in Figure 5-4. The figure depicts a confrontation between the agent and a predator of type A.

The sequence of events within the Policy Selection Module when a predator of type A is perceived is as follows:

1. Insert state information into the policy mapping function. The type A predator policy is the output of this function given the state information.
2. Set the type A predator policy as the current active policy in memory.
3. Select the appropriate action given the type A predator policy in memory.

The predator type A policy is selected based on the conditions of a production rule being satisfied. In the above case, state information indicates that the humanoid is currently perceiving a predator of type A. This matches the conditions of the production rule governing the selection of the predator type A policy and so the policy

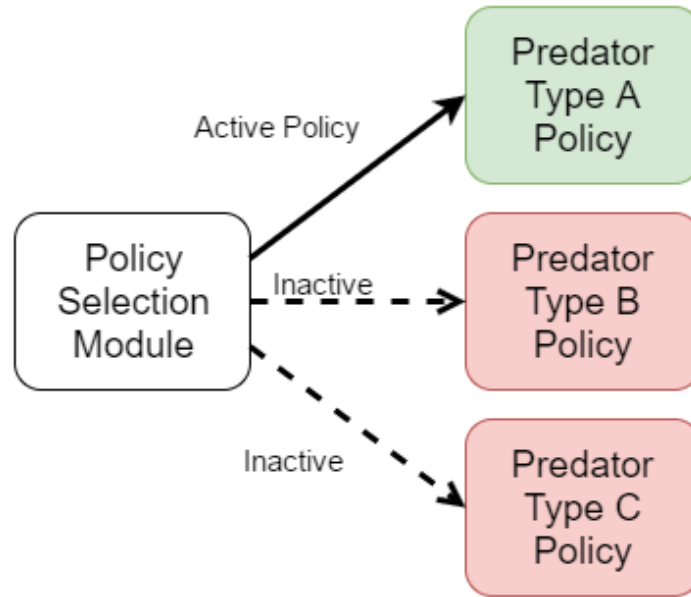


Figure 5-4: A depiction of the policy library containing the policies for predators A, B and C.

is output for selection.

Agents with and without policy selection were evaluated using the three parameter sets. The predator type count parameter is varied between parameter sets in order to demand a higher level of adaptable behaviour from agents. The average success of each agent and the average survival time is then compared for each parameter set in Figures 5-5 and 5-6. The average success and average survival time of an agent is calculated over the 60 runs in a simulation.

These results indicate that the policy selection mechanism allows the agent to adapt to an increase in predator types to a significantly greater extent than the agent without the mechanism. As the number of predator types increase, both agents display a drop in performance however the agent utilizing policy selection reveals a lower performance drop and a significantly higher average performance throughout experimentation. The reason for this difference in performance is due to the fact that the agent utilizing policy selection is able to maintain specialized policies for each predator type. Consider the policies generated by each agent for parameter set 3 as depicted in Figures 5-7, 5-8, 5-9 and 5-10. Each policy contains desired actions to

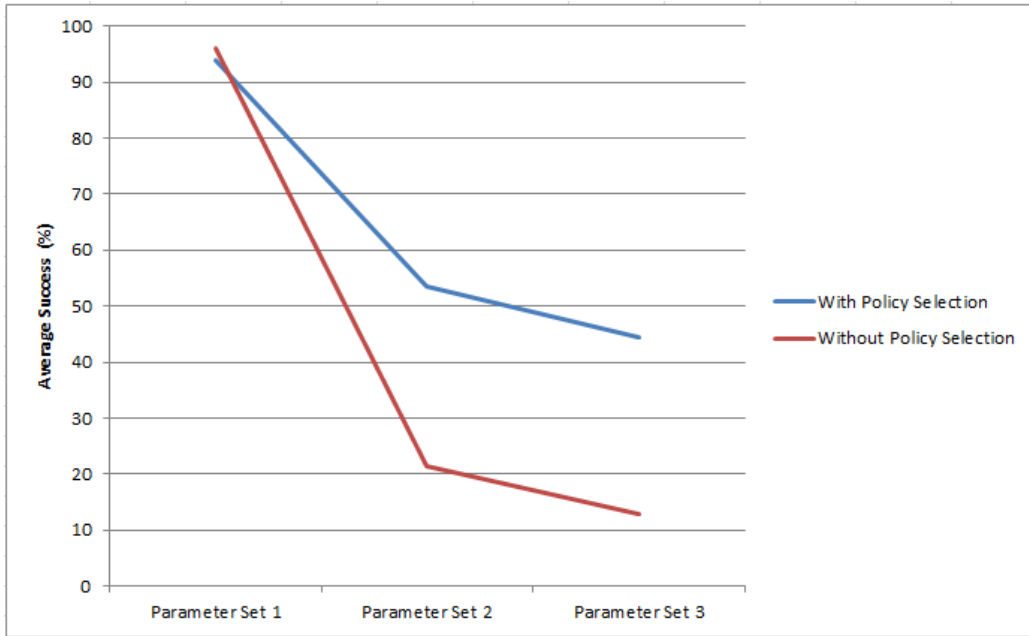


Figure 5-5: A figure depicting the average success of each agent utilizing the three parameter sets.

select given each environment state listed in Table 5.2.

During execution in parameter set 3, the agent utilizing policy selection was able to develop and refine three policies corresponding to predator types A, B and C respectively. This then allowed the agent to utilize the correct policy when dealing with each type of predator. The generic nature of the Policy Selection Module allowed the agent to automatically identify the need for three separate policies via predator attribute information. Whenever a new type of predator type was identified, the agent developed a new policy and stored it in the policy library. For example: When the agent encountered predator type A for the first time, a new policy was generated and inserted into the library. Once predator type B was identified via environment observations, another policy was then generated and utilized. The policies developed by the agent for facing predators A, B and C are depicted in Figures 5-8, 5-9 and 5-10 respectively.

Agent policies will yield the same results if the optimal actions for each state are equal in both policies. The policy developed in parameter set 3 by the agent without policy selection in Figure 5-7 matches the policy generated by the agent with policy

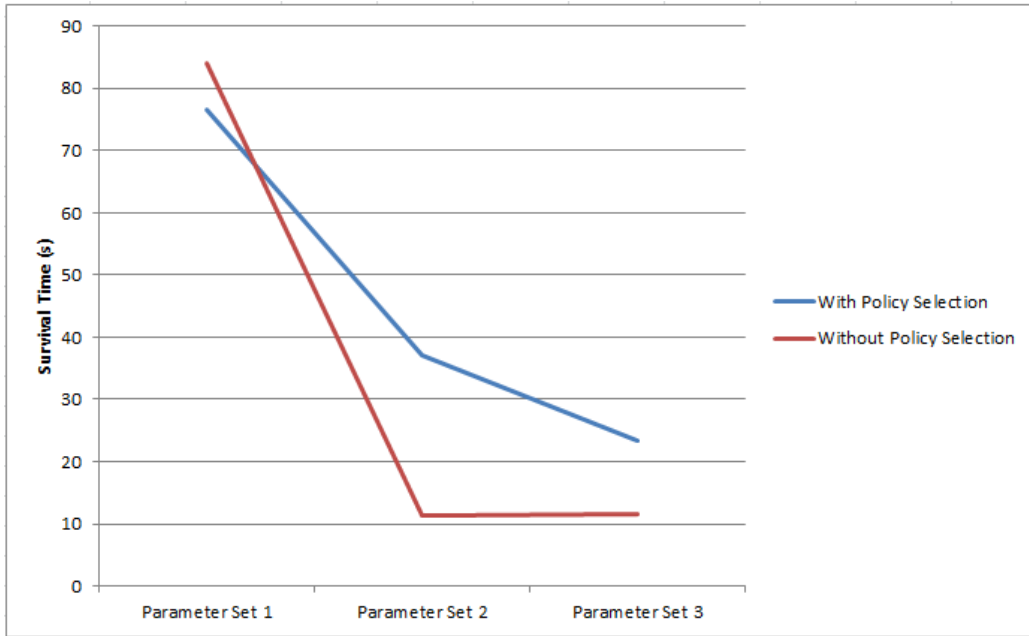


Figure 5-6: A figure depicting the average survival time of each agent utilizing the three parameter sets.

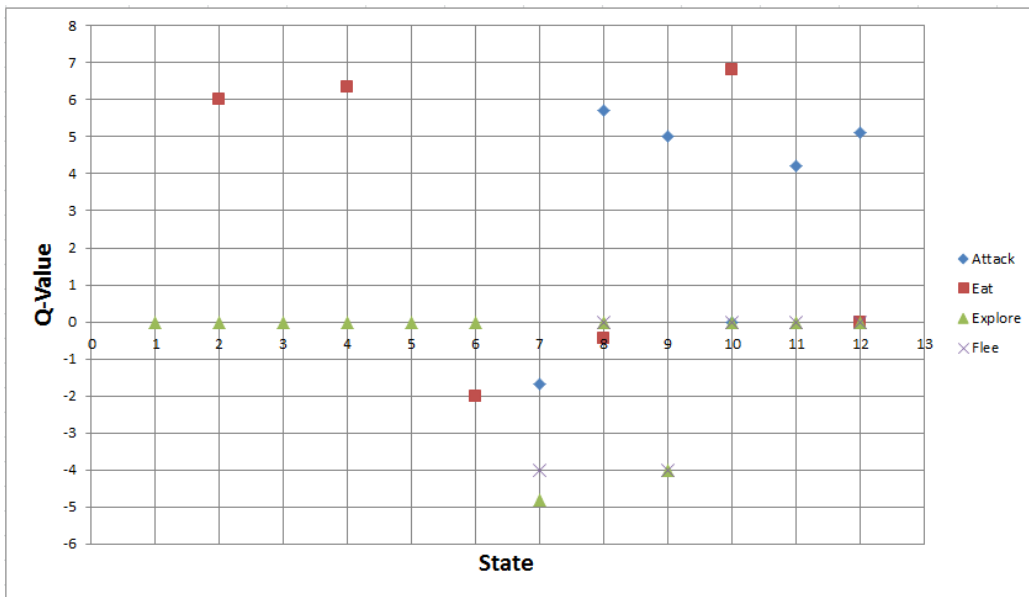


Figure 5-7: The single policy of the agent without policy selection in parameter set 3.

selection for predator type A in Figure 5-8. This indicates that the agent without policy selection was only able to account for a single predator type in the environment and so would result in a far lower success rate. Consider Figure 5-7, relating to a

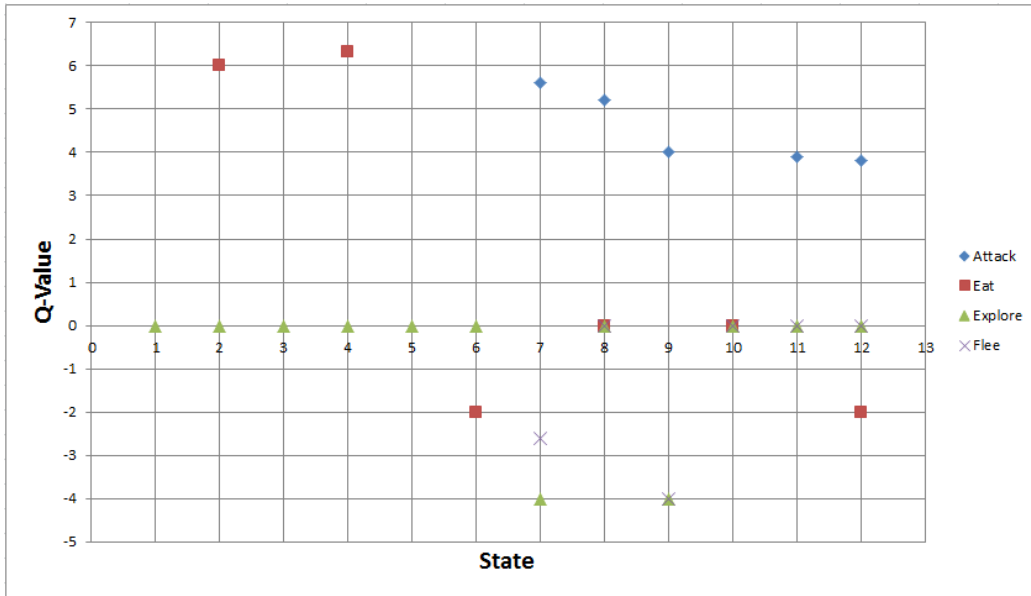


Figure 5-8: The developed policy for the agent against predator type A as stored in the policy library in parameter set 3.

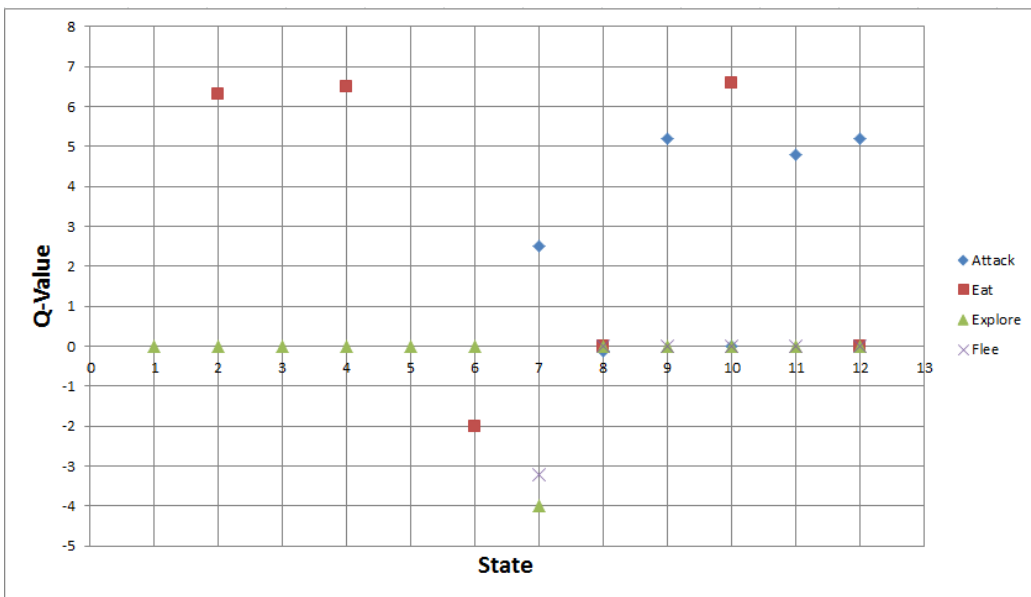


Figure 5-9: The developed policy for the agent against predator type B as stored in the policy library in parameter set 3.

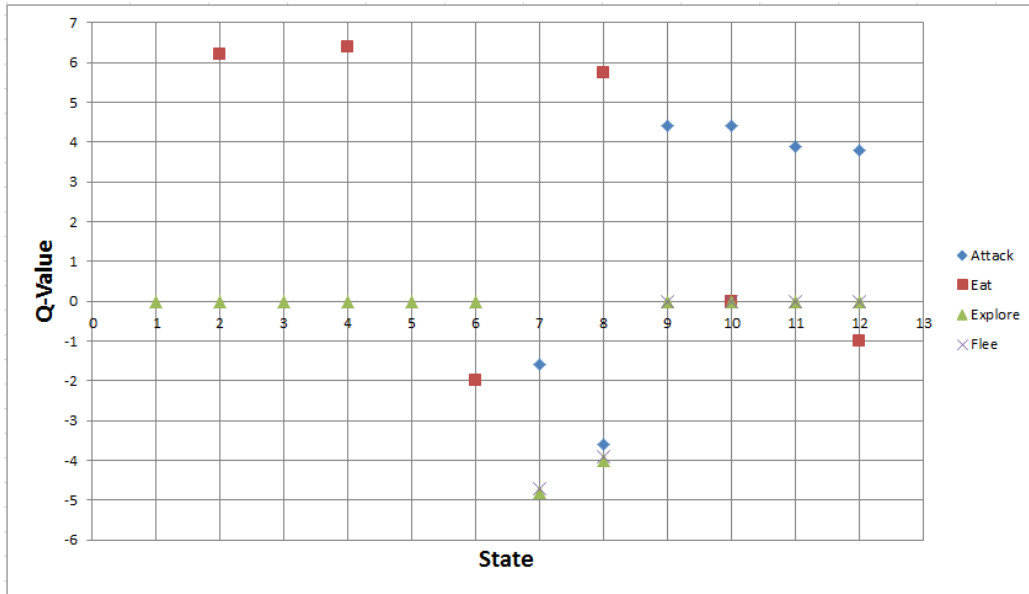


Figure 5-10: The developed policy for the agent against predator type C as stored in the policy library in parameter set 3.

policy developed by an agent without policy selection and Figure 5-10, relating to a policy developed by an agent with policy selection to account for stronger predators (Type C). Consider the proposed action in each of the policies for state number eight which correlates to a high danger level, a low health value and the availability of food being true. Regardless of surroundings and predator types, the agent without policy selection will always choose to engage nearby predators. If it was assumed that there are strong predators nearby (Type C), the agent utilizing policy selection would decide to seek food. The result of this defensive approach in this scenario would be improved survival time and a higher chance of survival in combat against the stronger predators. The agent not utilizing policy selection would be eliminated by any predators stronger than type A. This crucial difference in decision making results in the far lower success rate without the policy selection mechanism.

Figure 5-6 provides insight as to the average duration agents are able to survive in the world. On average, the agent utilizing policy selection is able to survive for a substantially longer period of time with a total average survival time of 46 seconds over the three parameter sets versus the 35 second average of the other agent. This indicates that the agent contains a more efficient method of adapting to the environ-

ment as it is able to survive for longer periods of time and achieve the given task of eliminating predators to a greater extent.

5.5 Experimental Platform Evaluation

The experimental platform was designed to be reusable by supplying extensible sensory, actuator and simulation support. Although it may be relatively easy to extend the existing support, should developers require a completely unique sensory or actuator system, such as an image capture sensor, this functionality would need to be integrated separately. An advantage of the current platform architecture does however allow developers to easily integrate new functionality with the existing system. If a new sensor mechanism was required, developers would simply need to replace the current sensory system and forward the corresponding sensor output to the relevant platform components.

5.5.1 The Q-Cog Application

The experimental platform was utilized in this work to evaluate the Q-Cog architecture. A scenario involving a humanoid agent with the goal of surviving in a virtual 3D world was created and implemented. The platform allowed for various simulations to be performed and the necessary results obtained. The Q-Cog application provides supporting evidence that the experimental platform may successfully be used to evaluate agent and cognitive agent architectures in 3D virtual environments.

5.5.2 Other Applications

Apart from the work done with the platform to evaluate the Q-Cog architecture, the platform was successfully utilized in three different research projects, two of which have been elaborated on below.

Deep Learning vs Shallow Learning in Support Decision Making in a 3D Virtual World

The research done by Ikram [16] aimed to compare the performance of deep and shallow neural networks for decision making in 3D environments. Two different agents were developed: One was trained with a shallow neural network while the other used a deep neural network. The experimental platform was utilized in this research to create scenarios whereby a humanoid agent would attempt to survive against hostile agents in a 3D environment. The combat extension was used to achieve the desired predator-prey system. The survival rate of the humanoid and agent eliminations was used as evaluation metrics. Figure 5-11 depicts cumulative agent kills obtained for the best performing shallow and deep learning agents in one experiment. The simulation engine of the experimental platform allowed for retrieval of the required metric data at appropriate events. Using these results from the platform, it was concluded that the supervised shallow learning network performs better than the deep learning network in the given scenario.

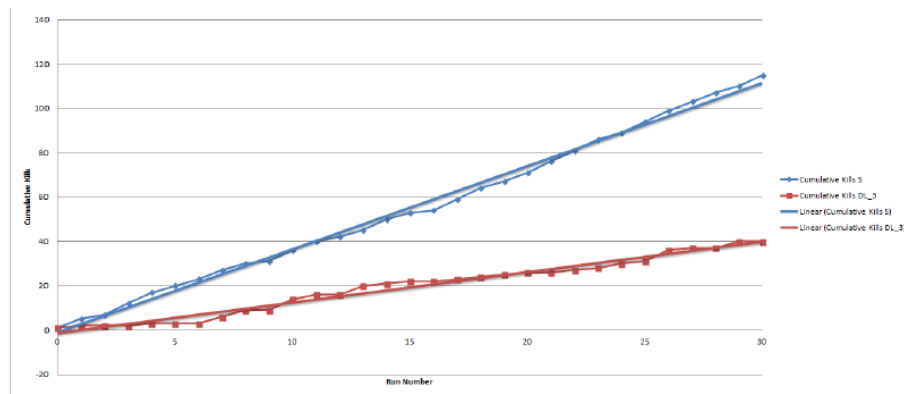


Figure 5-11: Cumulative agent kills based on results obtained by Ikram [16].

Ikram made the decision to utilize the architecture connector in order to implement back-end agent code in the Java language. All neural networks were then implemented separately from the experimental platform and simply utilized the connector to issue agent instructions and receive data.

A Comparison of Deliberative vs Reactive Agent Architectures in a 3D Virtual Environment

Research done by Ballim [2] aimed to compare various agent architectures within the 3D virtual world domain. Ballim evaluated deliberate and reactive architectures using the experimental platform. The research specified the following requirements for the evaluation platform:

- 3D sensory support.
- Multi-language support to utilize both the Java and C# programming languages.
- A stream of sensory data that may be analyzed by the architectures.

Ballim extended the platform to provide a hearing sensor as depicted in Figure 5-12. Each entity was modified to contain a hearing radius and objects in the world were given the capability to emit noise. This additional sensory data was then included in the sensory data stream to be processed by the architecture.

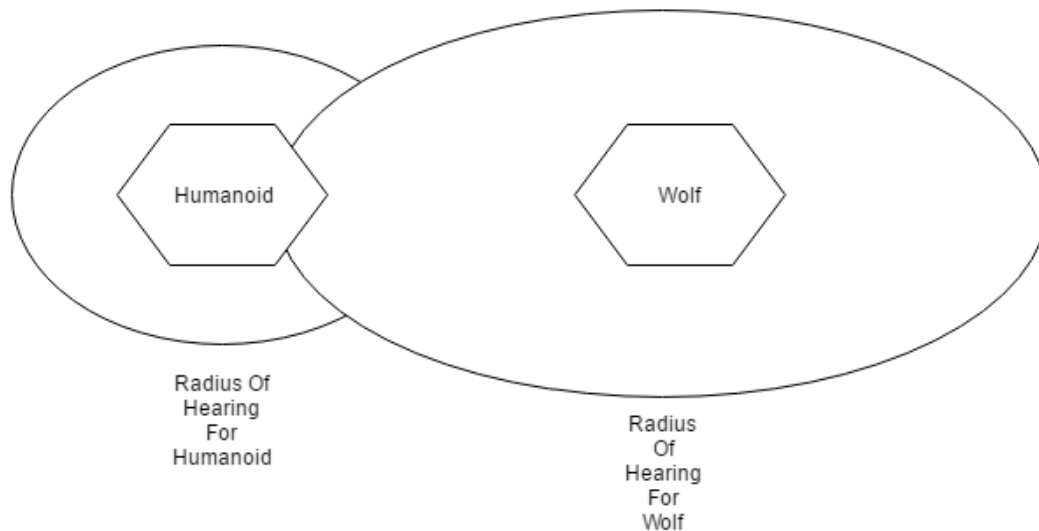


Figure 5-12: A depiction of the hearing sensor as implemented by Ballim [2].

Ballim created one scenario in the experimental platform which involved an agent maximizing the number of objects it can collect while avoiding other hostile agents in

the environment. Two experiments were successfully performed. The first experiment monitored agent survival time and the second experiment monitored the number of collected objects in order to evaluate agent performance. Figure 5-13 provides a section of results obtained from the platform by recording the amount of collected objects by agents with different architecture implementations. Based on the scenarios implemented, it was concluded that the BDI agent architecture outperformed other architecture implementations in the domain.

	0		1		2		3		4		5		6	
	BDI	REFLEX	BDI	REFLEX	BDI	REFLEX	BDI	REFLEX	BDI	REFLEX	BDI	REFLEX	BDI	REFLEX
1	21	6	11	13	13	17	16	12	9	6	8	6	7	11
2	10	16	15	15	14	16	22	13	11	6	4	3	19	5
3	15	14	14	17	13	10	9	13	14	6	7	5	8	7
4	20	16	16	18	13	10	8	10	13	14	5	5	10	17
5	18	17	12	12	14	7	14	13	9	5	20	6	13	8
6	18	10	11	13	8	4	21	8	16	4	9	10	14	7
7	20	18	18	19	12	13	8	17	14	12	14	5	8	12
8	16	16	19	13	6	5	7	8	17	6	14	7	11	2
9	14	12	9	18	13	16	16	8	15	10	9	8	12	9
10	16	7	11	9	8	20	6	16	7	15	8	8	10	5
11	18	15	10	20	9	17	18	9	15	5	5	8	19	15
12	12	14	16	11	13	19	10	16	9	14	11	5	13	9
13	14	17	11	12	13	12	13	11	7	7	6	5	20	6
14	11	17	10	18	13	5	6	10	15	6	12	7	13	12
15	9	14	14	18	21	9	14	10	7	8	7	10	13	5
16	14	11	12	13	7	7	7	17	5	8	9	4	12	13
17	14	20	10	20	10	10	10	17	7	13	5	6	7	6
18	12	13	13	14	18	18	10	13	14	15	10	7	13	4
19	17	15	21	10	16	11	15	13	10	11	7	5	11	7
20	14	19	17	19	13	12	13	12	13	5	5	11	15	9

Figure 5-13: Results obtained by Ballim relating to agent object collection for different architecture implementations [2].

5.5.3 Experimental Platform Results Summary

The evaluation of the experimental platform indicate that it may effectively be used to evaluate agents in a 3D virtual environment. Ballim [2] and Ikram [16] utilized the platform and were able to use provided support and easily add extensions in order to satisfy their experimentation requirements.

5.6 Summary

This chapter detailed the evaluation of the Q-Cog architecture and the experimental platform. Using the humanoid scenario within the experimental platform, six exper-

iments were conducted to evaluate the architecture. Results from experimentation indicate that the policy selection mechanism in the Q-Cog architecture allows agents to adapt more effectively to changes in the environment than agents without the mechanism. Evaluation of the experimental platform indicated that it may be used for many areas of artificial intelligence research including: Cognitive Agent Architectures, Deliberative and Reactive Architectures, Deep Learning and Reinforcement Learning. Researchers were able to extend the platform to meet their experimental requirements.

Chapter 6

Discussion

This chapter discusses the proposed Q-Cog architecture, its implementation and its empirical evaluation. This chapter is divided into three main parts: Theoretical analysis of the Q-Cog architecture and policy selection mechanism, analyzing the re-usability of the experimental platform and the empirical evaluation of the Q-Cog architecture.

6.1 Analysis of the Q-Cog Architecture

Q-Cog was inspired by several commonly used cognitive architectures, mainly, ACT-R [12], SOAR [23] and CLARION [27]. Table 6.1 outlines the main similarities of Q-Cog with existing cognitive architectures. The coordination mechanism, learning mechanism and knowledge representation scheme were used as dimensions for the comparisons. These dimensions will be explained below.

6.1.1 System Design

Coordination

Both the SOAR and ACT-R cognitive architectures contain a central processing module that controls the flow of information. The central processing module within SOAR maintains the current state of the world and also stores a representation of the current

Table 6.1: A comparison of cognitive architectures.

Architecture	Coordination	Learning Mechanisms	Knowledge Representation
Q-Cog	A Central Processing Module to control inter-module communication and perform actions on the environment.	A policy library stores learned policies that may be dynamically added and executed at runtime. The Q-Learning algorithm is then utilized when selecting an action and refining policies.	Contains a mixed approach with production rules and reinforcement values.
ACT-R	The procedural module coordinates all other ACT-R modules. Other modules communicate with this module through buffers.	Production compilation is used to generate new productions by merging separate rules into one. These may then replace multiple rules and reduce execution time. Utility values are also used when selecting actions and are refined after action execution.	Contains a mixed approach with utility values and production rules.
SOAR/ SOAR-RL	Working memory contains current state information in the form of working memory elements. It also coordinates the execution of actions on the environment.	Chunking is utilized in order to generate new procedural knowledge. SOAR-RL introduced Reinforcement Learning into the architecture.	Largely a symbolic architecture with the introduction of SOAR-RL adding a numeric aspect.
CLARION	The meta-cognitive subsystem controls and manages other subsystems in the architecture.	Utilizes bottom-up and top-down learning. Each module is then split into a top and bottom level. The bottom level utilizes Q-Learning while the top level utilizes rule extraction.	The top level of a module contains symbolic production rules with the bottom level containing a numeric representation.

goal of the agent. This module serves a similar purpose within the ACT-R architecture which is to provide a mode of communication between all other modules. Each module in ACT-R must utilize the procedural module in order to communicate with other modules. The central processing in Q-Cog was designed using principles similar to that of ACT-R and SOAR as the Central Processing Module controls the flow of information between all other modules. This allows tight control over the flow of information within the architecture.

Knowledge Representation

The SOAR architecture is predominantly symbolic. Knowledge is largely stored in symbols which function as a reference to an object or a rule. For example, a symbol may represent a real world object such as a tree. SOAR has recently included numeric representations, such as reinforcement and utility values, via the introduction of SOAR-RL. Numeric knowledge is utilized in SOAR when symbolic knowledge is not sufficient to make a decision. Both ACT-R and CLARION contain a mix of both

symbolic and numeric representations. In particular, CLARION divides each module into a symbolic and non-symbolic section due to its top-down and bottom-up learning approach.

Q-Cog has also taken a mixed approach to knowledge representation with the main focus on numeric representations. Numeric data is used when deciding on an action to select and symbolic data is used to constrain the actions in order for a subset of actions to be proposed. Production rules stored in production memory are used along with the Q-Learning Module when selecting actions. This approach has been utilized in both ACT-R [37] and CLARION [27] with the SOAR-RL architecture moving in the same direction [30]. A mixed approach reduces the amount of uncertainty in decision making as was found in the SOAR architecture [30]. When deciding on an action to perform, the architecture has access to both numeric and symbolic preference data. When symbolic data is not sufficient to make a decision, numeric knowledge may be utilized and vice-versa. This approach is more flexible.

Learning and Action Selection

The SOAR architecture selects a particular operator to execute on every decision cycle. Since SOAR uses a state-based representation, an operator modifies the current state in some way. Each operator is assigned a preference indication that provides a utility that operators use to compete for selection based on the current state. The operator with the highest preference is selected for execution. If two operators contain equal preferences, SOAR enters what is known as an *impasse* to solve this. When this impasse occurs, SOAR uses it as a learning procedure. The process known as chunking takes a "snapshot" of the current contents of working memory. This snapshot is then used to generate a new production rule that will eliminate the need for an impasse if a similar situation ever arises again. The recent introduction of SOAR-RL added a Reinforcement Learning aspect to the original architecture [30]. Operator preferences are now updated by including reinforcement values from the environment.

ACT-R utilizes a learning mechanism known as production compilation. Production compilation is a deductive process that involves the creation of new rules. These new rules are created to handle more specific tasks. These rules may then be used in the future to save processing time as they would replace the more general rules. Similar to the operator preference values in SOAR, ACT-R uses rule utility values where the utility of a rule is based on its success rate [11, 12].

The CLARION architecture uses a combination of bottom-up and top-down learning [21, 27, 12]. This involves the symbolic modules at the top level learning information by utilizing the numeric bottom level and vice-versa. Bottom-up learning is used to generate new rules. Top-down learning involves training the bottom-level neural network using data from the top level.

The goal of learning in Q-Cog is to allow agents to effectively adapt to changes in the environment. Reinforcement provides the means for agents to learn using their own experiences and hence environment changes will be taken into consideration. Q-Cog learning allows a virtual agent to develop a number of policies to adapt and perform optimally within a changing environment utilizing the policy selection mechanism. For this reason the Q-Learning and Policy Selection Modules were integrated into the architecture as the central learning modules. The focus of the learning process in Q-Cog is to develop multiple policies that the agent may utilize throughout execution as opposed to a single policy in traditional Q-Learning. Memory aspects such as episodic memory, commonly found in cognitive architectures, are important in order to achieve this due to the fact that agents require a means of storing important data.

Policy Selection

Other cognitive architectures such as CLARION utilize the Q-Learning algorithm but they do not maintain a policy library such as that in Q-Cog. The integration of the policy library in Q-Cog brings potential for cognitive architecture to focus on a dif-

ferent area of learning in the form of context-switching via policies [5]. The dynamic policy generation mechanism in Q-Cog allows the agent to maintain a library of specialized policies for different areas and situations within the environment as opposed to attempting to develop a single general policy that may be ineffective. Knowledge gathered in the policy library from previous iterations is stored and utilized in further iterations which reduces the need to start the learning process again. Consider the example of an agent acting within the Malmo platform outlined in Section 2.4.1. A standard agent in this environment may be required to navigate the world while constructing various objects or structures. The policy library proposed in Q-Cog allows for the world navigation and object construction to be separated into different policies as opposed to having a single general policy. Further separation may then occur within both the navigation and object construction policy subsets to develop specialized policies for different activity sets. This mechanism allows Q-Cog agents to maintain a high level of adaptability and flexibility. When agents encounter new portions of the state space, they are able to either use existing policies or generate a new policy to better adapt to the changes.

Empirical Evaluation of Q-Cog

The results of experimentation show that the Q-Cog architecture increases the adaptability of agents. Utilizing policy selection, agents are able to maintain specialized policies improving adaptability when the environment changes. Results were obtained for agents with and without policy selection i.e. agents A and B respectively. A comparison of the results obtained for the two agents in Figures 5-5 and 5-6 indicate a substantial performance improvement due to the utilization of policy selection. The performance difference is mostly evident when the number of predator types in the environment is increased. As the predator types increase, the need for adaptability becomes more important as agents must account for varying predator behaviour. Agent A is able to adapt to the increase in predator types to a far greater extent than agent B. This shows that the policy selection mechanism allows Q-Cog agents to adapt more effectively to environmental changes. The results obtained thus indi-

cate that the selection mechanism improves the performance of the agent within the given scenario as agent B was unable to successfully adapt to environment changes.

6.2 The Policy Library

6.2.1 Design and Integration

The role of policy selection in Q-Cog is to allow the agent to have specialized policies for different aspects of the environment. The policy selection mechanism was integrated into the architecture through the addition of a new module which maintains a policy library. An optimal policy contains optimal actions for an agent to perform in each environment state. Q-Cog generates and refines policies at runtime while continuously selecting the best policy to execute at a given time-step. This selection mechanism utilizes agent observations from the environment in order to correctly select the best policy. Agent observations are inserted into a function that maps environment states to policies and the relevant policy is retrieved for execution. As seen in work done by Lample and Chaplot [24], the complex tasks of navigation and combat were split into two policies and handled separately. Q-Cog allows for any number of policies to be generated and stored within the policy library to handle specific aspects of the environment such as: Navigation, combat and item collection. As an example, consider again the case of a humanoid agent surviving in a world containing hostile predators. These predators may vary drastically in behaviour to the extent that a single policy is not sufficient to allow the agent to survive in the environment. Once environment states have been formed, the agent may then autonomously develop a mapping of situations to policies which would allow for adaptation to each type of predator behaviour by selecting the appropriate policy. The potential of this mechanism may include the agent autonomously developing the map without the need for additional observation data to be inserted by developers. One possible area of interest for future work may include implementing deep learning for this purpose to allow the agent to autonomously identify unique aspects of the environment.

6.2.2 The Policy Selection Process

Section 2.2.3 outlined previous work on policy libraries by Fernandez and Veloso [10] and Chalmers et al [5]. The role of the policy selection mechanism in Q-Cog is to maintain a library of policies that each pertain to particular situations, i.e. a set of states, in the environment. The mechanism developed by Fernandez and Veloso probabilistically utilizes previous policies to assist in the learning of a new policy. The approach taken by Chalmers et al [5] extended the work done by Fernandez and Veloso. The mechanism involved context-switching whereby the agent maintained a library of learned policies and would select the best policy to execute given environment data. The agent checks the appropriateness of the current policy given recent state transition history and may either continue utilizing the current policy or retrieve a different policy from memory. The comparison with the design in Q-Cog hence lies in the selection process. Q-Cog does not utilize state transition history and instead utilizes a function that maps a set of environment states to a policy in order to select the most appropriate policy. The architecture generates this function automatically by analysing differences in environment observations. These differences are defined by the developer and utilized in the function. This may result in a more reliable policy selection scheme as developers may insert certain environment observations that may assist in the construction of this mapping function. An example of this can be seen in the results from experimentation where the agent was able to distinguish the three predator types from environment observation data and generate three separate policies.

6.2.3 Policy Selection in Game Environments

Previous work performed by Waltham and Moodley [46] investigated the addition of stochastic behaviour to agents in game environments. Although this provides less predictable opponents for human users, it does not successfully allow agents to adapt in order to counteract the strategy of opposing players. Agents in virtual worlds, and particularly in game environments, should be able to develop and modify a policy that

allows them to react to potentially drastic changes in the environment and human user actions. The policy selection mechanism in Q-Cog may be applied to a wide range of virtual world applications. Take for example a first-person shooter game similar to that used by Lample and Chaplot [24]. In this virtual world application, the agent has the task of competing against a number of human players. Each human player in the game will presumably have a unique style of play. It is thus difficult to generate a single generic strategy to effectively deal with all opposing players. Therefore, if the proposed policy selection mechanism was utilized, each time a new player is encountered by the agent in the world a new policy may be generated and maintained for dealing with that particular player. The agent would then simply have to select the appropriate policy when dealing with the corresponding player. This policy selection function would simply involve a mapping of policies to all players in the game and when a player is encountered, the appropriate policy is selected. The flexibility of the architecture allows for more strategies to be added as new players join the game.

6.3 The Experimental Platform

The experimental platform was developed and packaged to provide support for the integration and evaluation of agent-based research in 3D virtual worlds. The virtual nature of the platform provides researchers with a cost effective means of performing experimentation as opposed to utilizing physical devices such as robots. Experimentation in the real world brings about challenges in terms of controlling parameters. Virtual platforms allow the user to easily modify and control parameters such as simulated physical constants. This removes the need for expensive lab equipment and accelerates the development of an environment to perform experiments. This may however also reveal a limitation of utilizing virtual environments as they may not accurately mimic real world properties. The environment does provide a suitable emulation of real world properties such that researchers may gain insight as to how the work may function in a real world scenario. If the work has been found to be effective

through virtual world evaluation, it may validate the expense and time required to implement the work in real world scenarios such as robot interactions.

The research done using the platform involves many different areas of agent-based experimentation: Cognitive Agent Architectures, Deliberative and Reactive Architectures, Deep learning and Reinforcement learning. This shows that the platform does provide the necessary tools to allow for artificial intelligence based research to be performed within 3D virtual environments. The experimental platform provides an extensible, modular tool for developing and evaluating virtual 3D agents. Additional features may be added to craft ideal scenarios for different areas of agent-based research in virtual environments.

Chapter 7

Conclusion and Future Work

The objectives of the research were as follows:

- Design a cognitive agent architecture that allows 3D virtual agents to adapt to a changing environment using concepts from state-of-the-art architectures.
- Develop a virtual 3D experimental platform that may be reused in future agent-based research. Design and implement a complex scenario within this platform to evaluate the architecture.
- Integrate a policy library into the architecture to allow agents to adapt to drastic environment changes.
- Evaluate the architecture.

This research proposes Q-Cog: A Q-Learning based cognitive agent architecture for adaptive 3D virtual agents. An experimental platform was designed and implemented to evaluate the Q-Cog architecture. Various scenarios were designed within the experimental platform and experiments were performed. The architecture used key concepts of state of the art architectures such as CLARION, ACT-R and SOAR.

A complex humanoid scenario was designed and implemented within the experimental platform in order to test the effectiveness of the architecture in various scenarios. A major outcome of experiments includes validating the adaptiveness of the

architecture as a result of the proposed policy selection mechanism. The proposed scenario, involving a humanoid attempting to survive in a simulated virtual world, provides various complexities unique to the domain, such as, 3D world navigation. A concrete implementation of the Q-Cog architecture was developed in order to control the behaviour of the humanoid agent.

The architecture, when used without policy selection, was found to perform well in simple scenarios in the environment and was able to achieve a high average success rate over the course of 60 iterations. As the complexity of the scenarios was increased, the performance of the architecture without the policy selection mechanism rapidly decreased. The performance of an agent utilizing policy selection was significantly better for scenarios of higher complexity and outperformed the agent without the mechanism. Comparisons of results over a number of simulations indicated that the policy selection mechanism in Q-Cog successfully improves the adaptability of agents within the domain.

The experimental platform holds much potential for artificial intelligence research in virtual worlds. The platform was used successfully in a number of artificial intelligence areas such as: Cognitive Agent Architectures, Deliberative and Reactive Architectures, Deep Learning and Reinforcement Learning. This platform may be extended in future work to support a wider variety of scenarios that developers may implement for various artificial intelligence requirements.

In summary, the proposed policy selection mechanism in the Q-Cog architecture has been shown to improve the adaptability of agents in the 3D virtual world domain. The developed experimental platform assisted the experimentation performed in this work. Results gained indicate that the platform may successfully be used to perform agent-based research in the virtual world domain.

7.1 Future Work

- Future implementations of the Q-Cog architecture may include production compilation techniques similar to that in the ACT-R architecture in order to improve the policy selection mechanism. This may be used to autonomously generate policy selection rules in addition to predefined rules by developers.
- One major application of the proposed architecture and RL includes 3D video games. Future work may therefore include evaluating this architecture within a complex video game environment and comparing results obtained with previous work done by Waltham and Moodley [46].

Bibliography

- [1] Majed Alhajry, Faisal Alvi, and Moataz Ahmed. TD (λ) and Q-Learning Based Ludo Players. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, pages 83–90. IEEE, 2012.
- [2] M. Ballim. A Comparison of Deliberative vs Reactive Agent Architectures in a 3 Dimensional Virtual Environment [unpublished thesis]. 2016.
- [3] Trevor Barron, Matthew Whitehead, and Alan Yeung. Deep Reinforcement Learning in a 3-D Blockworld Environment. *Deep Reinforcement Learning: Frontiers and Challenges, IJCAI*, 2016:16, 2016.
- [4] Sarah Calderwood, Kevin McAreavey, Weiru Liu, and Jun Hong. Uncertain Information Combination for Decision Making in Smart Grid BDI Agent Systems. *International Journal of Industrial Control Systems Security*, 1(1):21–30, 2016.
- [5] Eric Chalmers, Edgar Bermudez Contreras, Brandon Robertson, Artur Luczak, and Aaron Gruber. Context-Switching and Adaptation: Brain-Inspired Mechanisms for Handling Environmental Changes. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 3522–3529. IEEE, 2016.
- [6] S. T. Cheng, T. Y. Chang, and C. W. Hsu. A Framework of an Agent Planning with Reinforcement Learning for E-Pet. In *International Conference on Orange Technologies (ICOT)*, pages 310–313, March 2013.
- [7] Hui-Qing Chong, Ah-Hwee Tan, and Gee-Wah Ng. Integrated Cognitive Architectures: A Survey. *The Artificial Intelligence Review*, 28(2):103, 2007.
- [8] John David N. Dionisio, William G. Burns III, and Richard Gilbert. 3D Virtual Worlds and the Metaverse: Current Status and Future Possibilities. *ACM Comput. Surv.*, 45(3):34:1–34:38, July 2013.
- [9] Fernando Fernández and Manuela Veloso. Building a Library of Policies through Policy Reuse. Technical report, DTIC Document, 2005.
- [10] Fernando Fernández and Manuela Veloso. Probabilistic Policy Reuse in a Reinforcement Learning Agent. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 720–727. ACM, 2006.

- [11] Michael E. Hansen, Andrew Lumsdaine, and Robert L. Goldstone. Cognitive Architectures: A Way Forward for the Psychology of Programming. In *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*, pages 27–38. ACM, 2012.
- [12] Sebastien Helie and Ron Sun. Autonomous Learning in Psychologically-Oriented Cognitive Architectures: A Survey. *New Ideas in Psychology*, 34:37–55, August 2014.
- [13] S. M. Hung and S. N. Givigi. A Q-Learning Approach to Flocking With UAVs in a Stochastic Environment. *IEEE Transactions on Cybernetics*, PP(99):1–12, 2016.
- [14] K. S. Hwang, W. C. Jiang, and Y. J. Chen. Tree-Based Dyna-Q Agent. In *2012 IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM)*, pages 1077–1080, July 2012.
- [15] K. S. Hwang, W. C. Jiang, and Y. J. Chen. Model Learning and Knowledge Sharing for a Multiagent System With Dyna-Q Learning. *IEEE Transactions on Cybernetics*, 45(5):978–990, May 2015.
- [16] M. Ikram. Deep Learning Versus Shallow Learning In Supporting Decision Making In A 3D Virtual World [unpublished thesis]. 2016.
- [17] O. Javier and R. Lopez. Self-Organized and Evolvable Cognitive Architecture for Intelligent Agents and Multi-Agent Systems. In *2010 Second International Conference on Computer Engineering and Applications (ICCEA)*, volume 1, pages 417–421, March 2010.
- [18] Seongsik Jo, Rohae Myung, and Daesub Yoon. Quantitative Prediction of Mental Workload with the ACT-R Cognitive Architecture. *International Journal of Industrial Ergonomics*, 42(4):359–370, July 2012.
- [19] Matthew Johnson, Katja Hofmann, Tim Hutton, and David Bignell. The Malmo Platform for Artificial Intelligence Experimentation. In *International joint conference on artificial intelligence (IJCAI)*, 2016.
- [20] Y. Kang and A. H. Tan. Self-Organizing Agents for Reinforcement Learning in Virtual Worlds. In *The 2010 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, July 2010.
- [21] Mohammad Kazemifard, Nasser Ghasem-Aghaee, and Tuncer I. Oren. Emotive and Cognitive Simulations by Agents: Roles of Three Levels of Information Processing. *Cognitive Systems Research*, 13(1):24–38, March 2012.
- [22] John E. Laird. Extending the SOAR Cognitive Architecture. *Frontiers in Artificial Intelligence and Applications*, 171:224, 2008.

- [23] John E. Laird, Allen Newell, and Paul S. Rosenbloom. SOAR: An Architecture for General Intelligence. *Artificial intelligence*, 33(1):1–64, 1987.
- [24] Guillaume Lample and Devendra Singh Chaplot. Playing FPS Games with Deep Reinforcement Learning. In *AAAI*, pages 2140–2146, 2017.
- [25] Pat Langley, John E. Laird, and Seth Rogers. Cognitive Architectures: Research Issues and Challenges. *Cognitive Systems Research*, 10(2):141–160, June 2009.
- [26] Michael L. Littman. Reinforcement Learning Improves Behaviour from Evaluative Feedback. *Nature*, 521(7553):445–451, 2015.
- [27] Danilo Fernando Lucentini and Ricardo Ribeiro Gudwin. A Comparison Among Cognitive Architectures: A Theoretical Analysis. *Procedia Computer Science*, 71:56–61, 2015.
- [28] Koichiro Morihira, Tejiro Isokawa, Haruhiko Nishimura, and Nobuyuki Matsui. Characteristics of Flocking Behavior Model by Reinforcement Learning Scheme. In *2006 SICE-ICASE International Joint Conference*, pages 4551–4556. IEEE, 2006.
- [29] J. Muhammad and I.O. Bucak. An Improved Q-Learning Algorithm for an Autonomous Mobile Robot Navigation Problem. In *2013 International Conference on Technological Advances in Electrical, Electronics and Computer Engineering (TAECE)*, pages 239–243, May 2013.
- [30] Shelley Nason and John E Laird. SOAR-RL: Integrating Reinforcement Learning with SOAR. *Cognitive Systems Research*, 6(1):51–59, 2005.
- [31] Andrew Nuxoll, John E. Laird, and M. James. Comprehensive Working Memory Activation in SOAR. In *Proceedings of the international conference on cognitive modeling*, pages 226–230, 2004.
- [32] Andrew M. Nuxoll and John E. Laird. Enhancing Intelligent Agents with Episodic Memory. *Cognitive Systems Research*, 17:34–48, 2012.
- [33] M. Pouyan, A. Mousavi, S. Golzari, and A. Hatam. Improving the Performance of Q-learning Using Simultaneous Q-Values Updating. In *2014 International Congress on Technology, Communication and Knowledge (ICTCK)*, pages 1–6, November 2014.
- [34] Anand S Rao, Michael P Georgeff, et al. BDI Agents: From Theory to Practice. In *ICMAS*, volume 95, pages 312–319, 1995.
- [35] Gavin Rens and Deshendra Moodley. A Hybrid POMDP-BDI Agent Architecture With Online Stochastic Planning and Plan Caching. *Cognitive Systems Research*, 43:1–20, 2017.

- [36] Stuart Russel and Peter Norvig. *Artificial Intelligence a Modern Approach*. Pearson Education Inc, 2010.
- [37] Dario D. Salvucci and Frank J. Lee. Simple Cognitive Modeling in a Complex Cognitive Architecture. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '03, pages 265–272, New York, NY, USA, 2003. ACM.
- [38] Alexei V. Samsonovich. Emotional Biologically Inspired Cognitive Architecture. *Biologically Inspired Cognitive Architectures*, 6:109–125, October 2013.
- [39] Guy Shani, Joelle Pineau, and Robert Kaplow. A Survey of Point-Based POMDP Solvers. *Autonomous Agents and Multi-Agent Systems*, 27(1):1–51, 2013.
- [40] Dharendra Singh, Lin Padgham, and Brian Logan. Integrating BDI Agents with Agent-Based Simulation Platforms. *Autonomous Agents and Multi-Agent Systems*, 30(6):1050–1071, 2016.
- [41] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*, volume 1. MIT press Cambridge, 1998.
- [42] Matthew E. Taylor and Peter Stone. Transfer Learning for Reinforcement Learning Domains: A survey. *Journal of Machine Learning Research*, 10(Jul):1633–1685, 2009.
- [43] Unity Technologies. Unity3D. <https://unity3d.com/>. Accessed: 2017-12-20.
- [44] Jordi Vallverdú, Max Talanov, Salvatore Distefano, Manuel Mazzara, Alexander Tchitchigin, and Ildar Nurgaliev. A Cognitive Architecture for the Implementation of Emotions in Computing Systems. *Biologically Inspired Cognitive Architectures*, 15:34–40, 2016.
- [45] Joost van Oijen. *Cognitive Agents in Virtual Worlds: a Middleware Design Approach*. 2014.
- [46] Michael Waltham and Deshen Moodley. An Analysis of Artificial Intelligence Techniques in Multiplayer Online Battle Arena Game Environments. In *Proceedings of the Annual Conference of the South African Institute of Computer Scientists and Information Technologists*, page 45. ACM, 2016.
- [47] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.
- [48] Michael Wooldridge and Nicholas R. Jennings. Intelligent Agents: Theory and Practice. *The knowledge engineering review*, 10(02):115–152, 1995.
- [49] J. Zuters. Sequence Q-learning: A Memory-Based Method Towards Solving POMDP. In *2015 20th International Conference on Methods and Models in Automation and Robotics (MMAR)*, pages 495–500, August 2015.