BUG BATTLE ARTIFICIAL INTELLIGENCE

Controlling Software Agents with Dynamic Decision Networks

L'INTELLIGENCE ARTIFICIELLE DE LA BATAILLE DE BUG

Contrôle des agents logiciels avec les réseaux de décision dynamiques

A Thesis Submitted to the Division of Graduate Studies
of the Royal Military College of Canada
by

Bradley Matthew Rathbun
Captain

In Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Science

October 19, 2016

© This thesis may be used within the Department of National Defence
but copyright for open publication remains the property of the author.

To Astrid, Rory and Rhys, my thesis minions.

Acknowledgements

I would like to thank my supervisor, Dr Yawei Liang, for his guidance and unwavering confidence in my abilities. His dedication to the unrelenting questions, drafts and proof-reading were a critical component in completing this work.

I am grateful to the professors of the Department of Mathematics and Computer Science at the Royal Military College of Canada as well as many of the staff at the Canadian Army Command and Staff College and Directorate of Land Synthetic Environments who gave excellent advice and guidance while researching my initial topic of Agent-Based Models applied to combat modelling. It is with regret that the scope of implementation required to support these organizations forced a change of topic.

I am indebted to Dr François Rivest, Dr Mohan Chaudhry, Dr Bill Hurley and Dr Jack Brimberg for a myriad of discussions ranging from probability distributions through to genetic algorithms. Their academic mentoring was invaluable as I developed my research and concurrent lecturing duties.

Although there were occasional struggles balancing the demands of military duties and academic work, the support offered by Lieutenant-Colonel Ryan Walker and Major Kristopher Campbell were the catalyst needed for the final surge to complete experimentation and analysis.

My friends and colleagues Frank and Gina Decarie aided extensively with proof-reading and general academic or technical discussions.

Finally and most importantly, I offer my heartfelt appreciation to my wife Chelsea, who put up with countless hours of absence while we raised our young children. This work would not have been possible without her eternal patience and support.

Abstract

Agent-based models (ABMs) seek to predict behaviour or develop insights into a system by assessing emerging complex behaviour that results from individual entities adhering to simple rules. Partially observable Markov decision processes (POMDPs) can be used as an ABM for real-world problems where limited information is available to guide decisions and action outcome is variable. POMDPs are notoriously expensive to solve computationally, but Dynamic Decision Networks (DDNs) exploit independence in system variables to develop approximate solutions. Key features in a DDN are the reward and utility functions used to guide decisions made by the software agents. This research assesses the performance of a DDN-controlled agent against an agent designed with expert domain knowledge for an established simulation environment called "Bug Battle". Variations of reward and utility functions were tested to determine resulting differences in behaviour. It was found that employing DDNs was an effective strategy for agent performance.

Contents

Contents **v**

List of Tables **ix**

List of Figures **x**

1 Introduction **1**
1.1 Purpose 1
1.2 Background 1
1.3 Motivation 2
1.4 Hypothesis 2
1.5 Document Organization 2

2 Literature Review **4**
2.1 Introduction 4
2.2 Agent-based Models 4
2.2.1 Definitions 5
2.2.2 Physical Model 6
2.2.3 Behavioural Model 7
2.2.4 Environment 9
2.3 Learning Agents 10
2.3.1 Introduction 10
2.3.2 Markov Decision Process (MDP) 11
Core Concepts and Definitions 11
Partially Observable MDPs 13
Solving a MDP 14
2.3.3 Algorithm Review 14
Temporal-Difference Learning 14
Q-Learning 15
State-Action-Reward-State-Action (SARSA) 15

Monte Carlo Methods 15
2.3.4 Hierarchical Task Networks 15
2.3.5 Bayesian Networks and Extensions 16
Bayesian Network 16
Dynamic Bayesian Network 17
Decision Network 18
Dynamic Decision Network 18
2.4 Summary 19

3 Model Design **20**
3.1 Introduction 20
3.2 Common Agent Variants 20
3.2.1 Physical Model 21
Scout 21
Worker 21
Attacker 23
Feeder 23
Recce 23
3.3 Baseline Agent Design 23
3.3.1 Behavioural Model 25
3.4 Dynamic Agent Design 26
3.4.1 Reward and Utility Functions 27
3.4.2 DDN Implementation 28
3.4.3 Solving the DDN 30
3.5 Summary 33

4 Experiment Design **34**
4.1 Introduction 34
4.2 Parameter Selection Experiment Design 34
4.2.1 Preliminary Experiment - A Simplified Model 34
4.2.2 Parameter Selections 37
4.3 Distributed Environment 37
4.4 Summary 39

5 Results **40**
5.1 Introduction 40
5.2 Modified Experiment Design 40
5.3 Experiment Results 41
5.3.1 Data Collection 41
5.3.2 Processing Time 44

5.3.3 Win Percentages . . . 50
5.3.4 Strategy Selection . . . 51
Isolating Strategies . . . 51
Strategy Details . . . 54
5.3.5 Simulation Observations . . . 61
5.4 Summary . . . 62

6 Analysis 64
6.1 Introduction . . . 64
6.2 Dynamic Bug Performance . . . 64
6.2.1 Reward Function Implementation . . . 65
6.3 Computational Cost . . . 67
6.4 Assessing Agent Performance . . . 68
6.5 Summary . . . 69

7 Conclusion 70
7.1 Contributions . . . 70
7.2 Future Work . . . 71

A Glossary 72

B Bug Battle Simulation 74
B.1 Simulation Framework . . . 74
B.1.1 Simulation Logic . . . 74
Simulation Loop . . . 75
Simulation Combat . . . 78
B.2 Organs . . . 78
B.2.1 Budder . . . 80
B.2.2 Cilia . . . 80
B.2.3 Cloaking . . . 80
B.2.4 CreatureTypeSensor . . . 81
B.2.5 EnergySensor . . . 81
B.2.6 LifesignSensor . . . 81
B.2.7 PhotoGland . . . 81
B.2.8 PoisonGland . . . 82
B.2.9 PoisonSensor . . . 82
B.2.10 Spikes . . . 82
B.3 Techniques . . . 82
B.3.1 Probabilistic Exploration . . . 83
B.3.2 Frequency-Based Exploration . . . 84

B.3.3 Adaptive Spawning 85
B.3.4 Multiple Spawning 86
B.3.5 Energy Transfer 86
B.3.6 Kamikaze Strikes 87
B.3.7 Headquarters (HQ) 87
B.3.8 Scan Caching 88
B.3.9 Defeating Cloaking 89
B.3.10 Defensive Organs 89

C Particle Filtering and Decision Trees **92**
C.1 Particle Filtering 93
C.2 Decision Trees 95

D Supplementary Data **103**

Bibliography **104**

List of Tables

3.1 Scout Organ Build 22
3.2 Worker Organ Build 22
3.3 Baseline Movement Priorities 25
3.4 Baseline Spawning Priorities - Scout 25
3.5 Baseline Spawning Priorities - Worker 26
3.6 Dynamic Decision Network (DDN) Focus 30
3.7 DDN Action 31

4.1 Preliminary Experiment Results 36
4.2 Preliminary Experiment Summary 37

5.1 Permutation Win Rates 47
5.2 Dynamic Bug Performance - Reward and Utility Function Weights 51
5.3 Action Trends 63

B.1 Bug Battle Organs 79
B.2 Benefits of Scan-Caching (Worst-Case) 89

List of Figures

2.1 Summary of Behavioural Model 8
2.2 Annotated Block-like Representation of Interactive Components (BRIC) Model [32] 10

3.1 Probability of Spawning Baseline Worker Agents 24
3.2 Decision-Action Cycle for a Dynamic Bug Turn 27
3.3 Reward Sub-Functions 29
3.4 Building and Solving a Decision Tree - 1-step 32

4.1 Toy 3-State Model 35

5.1 Data Collection Database - Logical Design 43
5.2 Duration of Simulation Trials 44
5.3 Average Turn Duration 45
5.4 Average Trial Processing Time 48
5.5 Dynamic Bug Processing Time 49
5.6 Dynamic Bug Win Rates, by Parameter Permutation 50
5.7 Filtered Dynamic Bug Win Ratios, by Parameter Permutation . . 51
5.8 Average Bug Populations 52
5.9 Common Patterns for Bug Populations 53
5.10 Percentage of Trials Determined by the Leader 54
5.11 Trials When the Population Leader Won 55
5.12 Recovered Trials at Turn 10 55
5.13 Differences in Average Dynamic Bug Population 56
5.14 Typical Action Distribution on a Single Turn 57
5.15 Typical Action Distribution on a Single Turn - Filtered 58
5.16 Generalized Action Evolution 58
5.17 Comparative Action Evolution 59
5.18 Comparison between Permutations for Selected Actions 59
5.19 Progression of an Attack Action 60

B.1 Bug Selection 75
B.2 Simulation Execution 76
B.3 Simulation doTurn() Sequence Diagram 77
B.4 Ant-Colony Optimization (ACO)-Inspired Search Patterns 85
B.5 Magnification of the Attack Bug Damage Function 91
B.6 Attack Bug Damage Function 91

C.1 Toy 3-State Model 92
C.2 Particle Filtering Algorithm 95
C.3 Partial Decision Tree - Once Evidence is Generated 98
C.4 Full Decision Tree - with Details 100
C.5 Full Decision Tree - Abstracted 101
C.6 Full Decision Tree - Solved 102

1 Introduction

1.1 Purpose

This thesis presents a design to create an effective strategy to defeat adversaries in the Bug Battle simulation. The development of the strategy was based on machine learning techniques, guided by expert domain knowledge.

1.2 Background

Bug Battle is a simulation in which programmers design software creatures to which they can add organs and programmatically define behaviour. Once the simulation begins, competitors have no influence over their creatures. Bug Battle was developed by Dr. Greg Phillips and Major Gary Wolfman based on an original concept called BugWars developed by Dr. Scott Knight. The simulation is used to teach software engineering concepts in object-oriented analysis and design courses both at the undergraduate and graduate levels at the Royal Military College of Canada.

The goal of each simulation instance is to eliminate all competing creatures from the simulation environment. The environment is a physical representation of a world encoded as a 100 cell-by-100 cell world, represented as a square grid with horizontal and vertical wrapping. The simulation begins by placing a predefined number of each competing creature (called a bug) onto the world, which then has simulated plants added according to a uniform probability density. The simulation progresses by stepping through turns sequentially, in which each bug gets the opportunity to perform actions within the system rules. These energy-constrained actions are typically to add organs to increase capabilities and to employ the organs to gain dominance over the simulated world. Bugs are destroyed either by expending more energy than they have available, or via combat between bugs. Combat occurs when two bugs occupy the same cell; the bug having the greater energy becomes the victor. The

winning bug gains the energy from the losing bug. Losing bugs may impose defensive damage if they have added specialized organs. Bugs can multiply by asexual reproduction, which requires further energy expenditure.

Arguably the optimal strategy of Bug Battle is to optimize energy collection and use. The ways to gain energy include adding specialized organs at high initial cost or to attack plants or weaker bugs to gain their energy. A full review of Bug Battle is included at Annex B.

1.3 Motivation

The author participated in the original BugWars competitions as part of a credit for a graduate-level course. During development of the bug for these competitions, several extension points and methods to optimize performance were noted[1]. The aim of this thesis is to employ machine learning techniques to refine behaviour within a heuristically determined state space. Neither the number of turns required to achieve victory nor the computational cost are considered as important factors for the purpose of the aim, but they will be examined as the competition environment has a restriction for "reasonable" time limits.

1.4 Hypothesis

This thesis suggests that treating the Bug Battle simulation as a Partially Observable Markov Decision Process (POMDP) and solving the resulting Dynamic Decision Networks (DDNs) will yield improved performance compared to an expert design. A Baseline agent is developed from analysis of organ capabilities and system probabilities, as an extension of the most effective Bug Battle competitor to date. A Dynamic agent will have access to the same heuristically determined actions and will use a DDN to choose actions. The metric to assess agent effectiveness is the probability of winning a simulation trial with given reward and utility functions.

1.5 Document Organization

The thesis is organized as follows below. Each chapter ends with a summary of the material presented in the chapter.

[1] Optimization in terms of winning a simulation instance, not computational efficiency.

Chapter 2 is a literature review of agent-based models and reinforcement learning.

Chapter 3 develops the concepts for experimental bug models.

Chapter 4 describes the experiment design.

Chapter 5 presents the data gained from experiment trials.

Chapter 6 analyzes the results to determine the causes to gain insight into the design process and overall system.

Chapter 7 highlights the main findings and suggests future areas for research.

Annex A is the glossary for the document.

Annex B is an in-depth review of the Bug Battle system. Prior credited work in this area of research will be discussed here.

Annex C is a detailed example of particle filtering and solving decision trees.

Annex D is detailed data distributed via DVD. It contains the raw Structured Query Language (SQL) output of the simulation trials as well as SQL scripts to extract the data presented throughout the main body of the thesis.

2 Literature Review

2.1 Introduction

This chapter is divided into the following major sections and with a summary of the important items to be applied in Chapter 3.

- A review of agent-based modelling in general; and
- A review of learning systems used by learning agents, with a particular focus on reinforcement learning.

2.2 Agent-based Models

Agent-Based Model (ABM) follow a design philosophy of “bottom-up” modelling. They provide a mechanism for an experimenter to interact with low-level representations of entities, called agents, with the aim of gaining insight into the emergent behaviour that results from the plethora of agent interactions [5]. Some authors note this insight as being one purpose for such a model while prediction can be another [21]. The goal in this thesis is to gain insight to drive improved designs.

The literature on ABM has many authors who agree on core concepts, but with slight shifts in definitions or employment of the concepts to fit the particular situations that they model. To provide a consistent discussion, some definitions are provided in Section 2.2.1[1]. The main concepts of ABM will then be examined using these definitions.

Borshchev recognizes the lack of agreement of terminology in his comparison between ABM and system dynamics [4]. The central argument of his article is that ABM are decentralized. A consequence of this characteristic is that there is often a considerable amount of inter-agent communication,

[1]The definitions considered are not a comprehensive summary of characteristics or features within the scope of ABM. They are limited to those relevant to modelling Bug Battle due to scope constraints.

which translates to higher computational cost, if cooperative agents are desired. This is a major contributing factor to why ABM is a relatively new field; sufficient computational capacity is a recent development relative to the study of models in general. The size of the system under study can still cause computational difficulties, although there are techniques that may alleviate some of these issues. Based on the abstractions present in Bug Battle, this is a very relevant problem.

2.2.1 Definitions

Agents and their components lack crisp definitions in the literature. Russell and Norvig [20] provide a broad starting point which is "an agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors". Sometimes, it is appropriate to consider an agent to consist of a combination of a physical model and a behavioural (or mental) model. Both of these components utilize the idea of state, which in this case becomes a conditional variable that combines with perceptions to influence behaviour. Russell and Norvig further describe various sub-categories of agents that are differentiated by their behavioural sub-models. A table summarizing various agent capabilities (most of which are discussed below) is available at [22].

ABM research is normally focused on the simulated mental capacity of an agent. The mental capacity can be divided into two areas: state and processing. Shoham covers the justification for attributing mental terms to agents in his early work [23]. He suggests that agents are entities whose state consists of mental components such as beliefs, capabilities, choices, and commitments. To aid these mental components, other authors such as Macal and North [12] note that agents should have some form of memory to allow planning based on observations. Processing refers to how agents transform knowledge and observations into actions.

Autonomy is the ability of an agent to determine actions based on a combination of built-in knowledge and experience. This is derived from Russell and Norvig and others such as [36, 23, 9]. The requirement for agents to be autonomous is highly desirable across the literature as the sheer volume of agents, often in dynamic environments, precludes the ability for direct intervention by human operators or exhaustive descriptive rules.

Adaptability is the ability of an agent to explore a state space to make better decisions based on similar perceptions. This definition is based on Macal and North [12], among others. Reinforcement learning techniques (see Section 2.3.1) are inherently adaptable.

Beliefs can be considered as perceived knowledge [23]. For example, one of the organs in Bug Battle allows bugs to “cloak”, which renders the bug undetectable to sensors. When sensing the location of a cloaked bug, the sensing bug may believe that there is nothing there. The terminology and processes for this requirement vary, for example, see [32].

Capability refers to an agents capacity to effect the environment or other agents [23]. For example, in order to move a bug must have a cilia organ. Without one (or more) of this organ type, the bug is immobile.

Choice is synonymous with decision. At a given point in time, an agent will choose the selected action that it desires to perform from a set of potential actions based on capabilities and the mental assessment of perceptions.

Communication is the ability of agents to pass and receive messages among themselves. Agent communication can be affected by the environment, although this is not the case in Bug Battle. Communication does not have to be direct messages between bugs. A shared state is an example of an indirect communication method.

Reactivity is the ability to perceive the environment and initiate appropriate actions, which is a natural extension from the agent definition by Russell and Norvig [19]. The actual mechanism to achieve this may vary, with the simplest being a table of condition-action rules.

In situated environments the environment is a representation of spatial dispersion, physical terrain or other similar features. Ferber suggests that an ABM consists of an environment, agents, non-agent artifacts, relations between agents, and agent actions [9]. The environment will be examined in Section 2.2.4.

2.2.2 Physical Model

The physical model describes how an agent will interact with a simulated physical environment. Sensors provide the perception of the environment. The perceptions are assessed by the behavioural model, which results in the agent initiating actions. The actions may be physical or mental (or both simultaneously) and will depend on the capability of the agent.

Physical actions in the Bug Battle simulation include sensing, movement, cloaking, spitting poison and spawning. There are a variety of sensor organs that can be added and actuated to detect adjacent cells. Movement refers to an agent changing its position in the environment, which requires a cilia organ. Cloaking requires another organ and allows the agent to be undetectable to sensors. Spitting poison generates a notional agent on the targeted cell that causes defensive damage to an attacker via the combat mechanism. Spawning

occurs when an agent has a reproduction organ that can be actuated to create a new agent in an adjacent cell.

For consistency, once an agent has chosen an action to perform, the resolution of the action should be left to the environment [33]. This allows universal simulated physical laws to be reinforced. The organs employed in Bug Battle that can interact with the environment adhere to this model.

2.2.3 Behavioural Model

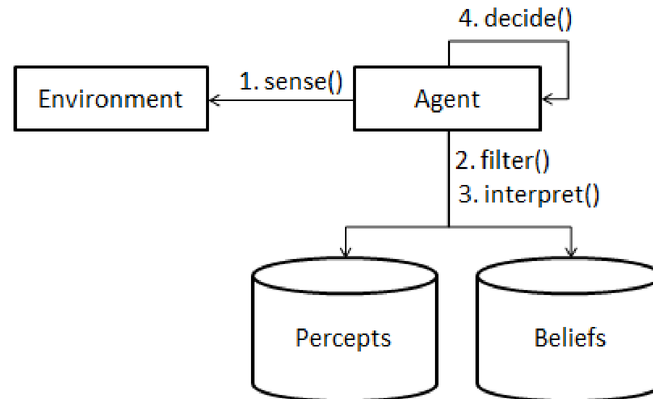
The behaviour model deals with the internal agent state and the mechanisms used to make decisions. Russell and Norvig [19] introduce mental state as remembering a fact, such as checking an adjacent lane before attempting to change lanes while driving. To perform the lane change safely the agent must first signal (a first action), then view the desired lane (a second action) and then move into the lane (a third action). Without the ability to retain the perception of the state of the lane, the third action could not reasonably be safely performed.

The mental model can further be augmented by knowledge or beliefs. Note that the example above implicitly assumes that another agent has not moved into the desired lane between the viewing and the movement[2]. Based on the serial nature of Bug Battle, the actions of other agents while a bug is executing a turn is not a factor. Beliefs in Bug Battle are expectations of the type of bug given an energy signature, or the probability of bugs beyond the local environment. See Section B.2.4 or Section B.3.7 for more information on beliefs in Bug Battle.

Percept is a term for information received by an agent from its sensors [19]. Individual or grouped segments of percepts allow a mapping of (potentially noisy) input to possible actions. Noisy sensor input would be input with randomized error from the environment applied.

Weyns et al proposed a generic model for active perception based on sensing, interpreting and filtering [32]. Sensing maps the environment state onto a representation influenced by perceptual laws (e.g. noise or hidden values determined by the environment). Interpretation maps a representation to a percept according to a description. The description may provide multiple layers of information, for example, a group of bugs might be recognized as a colony as well as being recognized as individual bugs. Filters are used to select only the percepts that are relevant for choosing an action.

[2] The associated "belief" here is that the probability of another vehicle moving into this space is exceptionally small, so it can be discounted.



This diagram shows how agents use information from the environment to make decisions.

1-2: The agent uses its sensors to obtain filtered information from the environment, which it maps onto a percept. The agent stores these percepts in its (potentially long-term) memory.

3: The agent assesses the memory of percepts and beliefs to generate potential value for the action(s) it has available.

4: The agent chooses the action that has the maximum expected utility.

Figure 2.1: Summary of Behavioural Model

Russell and Norvig stress the importance of assessing agent behaviour by rationality as opposed to omniscience [19]. A rational agent is defined as "one that does the right thing", which is dependent on a performance measure, precept history, environmental knowledge and capability. An example of a rational agent would be a bug moving onto a cell occupied by a cloaked bug. Unless the moving bug somehow suspected that cloaked bugs were present in the simulation, movement onto the apparently empty cell would be a rational decision.

Figure 2.1 summarizes the concepts presented in this section to show the abstract process normally used in the behavioural model[3].

[3]This figure was created for the thesis based on collaboration diagrams from the Unified Modelling Language to highlight the interaction between the environment and mental state of an agent.

2.2.4 Environment

An environment is the representation of the physical or relational world in a model. These representations may be very detailed or very abstract[4]. Weyns et al provide an extensive survey of ABM environments in [32]. They describe environments on a scale ranging from communicative (relational) to situated (physical). In a fully communicative environment, agents can only communicate, there is no spatial or action component. In a fully situated environment these properties are reversed. Since it is a scale, a wide range of hybrid environments are possible.

Weyns et al include a review of Ferber's Block-like Representation of Interactive Components (BRIC) model [9, 32]. An annotated version of the figure they presented is given in Figure 2.2[5]. This model is used to synchronize perception, actions, messages and effects between agents. The BRIC model effectively separates the action model from the agent representation, which is a key point in the later work of Weyns [33]. The motivation for this separation is to keep the design modular and ensure that all phases in a given cycle are synchronized.

The BRIC model is presented to demonstrate the importance of environmental modelling in agent-based models in general. Each version of Bug Battle does not directly adhere to this model when released[6], but it can be extended to provide most of the functionality present in the BRIC model. The deviations from the BRIC model are intentional given the development and purpose of Bug Battle[7]. The main differences are a serial turn scheduler instead of a synchronizer, and no message passing[8]. The Bug Battle simulation framework is discussed in depth at Annex B.

[4]Highly detailed environments are dependent on the system being modelled. For example, a simulation studying ant-colony optimization [6] may benefit from modelling the absorption rate of the terrain on the pheromones deposited by ants, as suggested in [33]

[5]The base model is from [32], it was annotated for this thesis to be able to be read out of context.

[6]Each year there are often minor revisions to correct minor problems in the software code.

[7]Bug Battle was developed as a teaching aid for introductory software engineering. The advanced design patterns required to implement the BRIC model are beyond the scope of the course for which it was designed.

[8]The intention of the synchronizer in the BRIC model is to collect multiple agent actions before executing agent actions in the environment whereas Bug Battle maintains a fixed environment while individual agents execute actions. Extending the Bug Battle framework to achieve message-passing can be achieved in a variety of ways such as use of the Observer pattern or by shared variables.

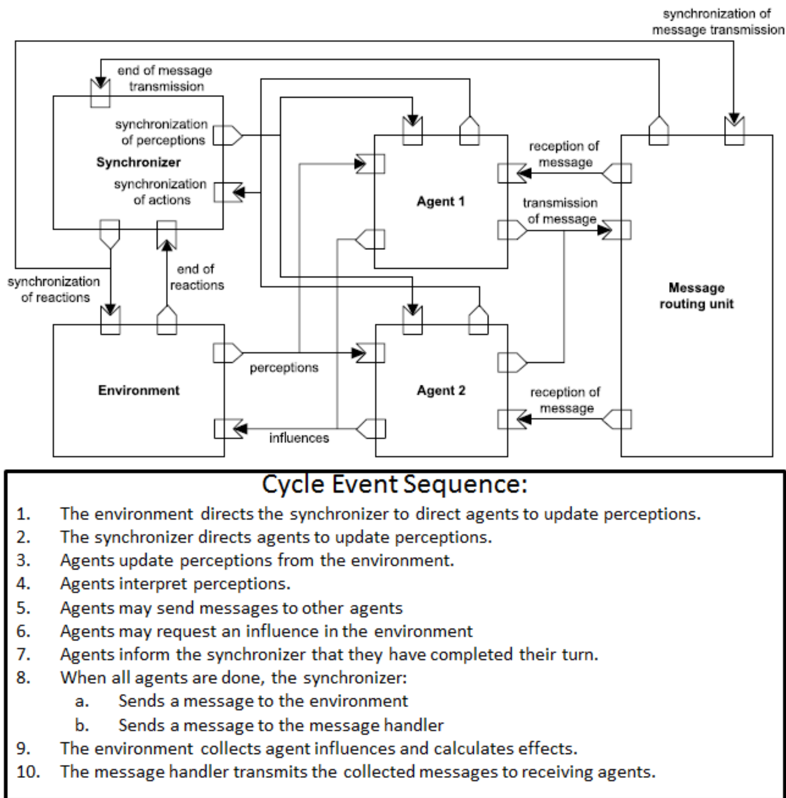


Figure 2.2: Annotated BRIC Model [32]

2.3 Learning Agents

2.3.1 Introduction

Learning agents are adaptive agents that apply some form of machine learning to their decision-making. There are three general approaches to machine learning methods: supervised learning, Reinforcement Learning (RL) and unsupervised learning [1, 20]. In supervised learning, the agent is supplied a training set of inputs and the associated value of the output. The goal is to learn a function that best maps the inputs to the given output so that future inputs can be similarly handled. In RL, the agent is given a reward function. The agent takes actions and receives rewards. The goal is to learn which actions to take when in a given state to maximize or minimize reward. In unsupervised learning, the agent is given inputs without any feedback and it must determine patterns.

Choosing which technique to use depends on the characteristics of the problem to solve. Russell and Norvig suggest that this choice should be based on what needs to be learned, prior knowledge available, representation of the problem and the feedback available to the agent [20]. The state space for Bug Battle is unnecessarily large and will need to be abstracted to some degree - how exactly this abstraction is chosen will be discussed in Chapter 3. Supervised learning is problematic for this domain as the number of (input, action) pairs is very large and the preferred outcome is challenging to appropriately assess. Despite this restriction, expert knowledge of the system can be used to generally guide preferred actions, which facilitates development of a reward function. RL is the approach chosen for this thesis.

2.3.2 Markov Decision Process (MDP)

Modelling real-world problems often involves representing the system as a series of choices that are dependent on previous choices. This complicates classical probability theory approaches as independence is usually assumed, which led to the development of Markov chains to allow this dependency based on the Markov assumption [16, 15]. The Markov assumption is that the current states depends on only a finite fixed number of previous states [20]. A first-order Markov process will rely on only the previous state, a second-order Markov process would rely on the previous two states, and so on. Markov processes are dependent on the Markov assumption to remain tractable.

Core Concepts and Definitions

Markov Decision Processs (MDPs) are used to model agent decisions when acting in a Markov process [20, 30, 1, 10]. Brief descriptions of the component parts are given below before formally presenting MDPs. These descriptions come primarily from [30], but the concepts are repeated throughout much of the literature.

The environment is defined as a finite set of states where $S = \{s_1, s_2, \ldots, s_n\}$. Any given state is a unique description of the environment components represented.

Actions are defined as a finite set where $A = \{a_1, a_2, \ldots, a_k\}$. Actions are used by agents to affect the state. Not all actions can be applied in each state. Two common ways to account for this are to use a pre-condition function $pre : S \times A \rightarrow \{true, false\}$ or to set the associated transition probability (see below) to zero.

The transition function is defined where an agent selects $a \epsilon A$ in state $s \epsilon S$, causing the system to transition to a new state $s' \epsilon S$. The choice of s' is determined by a probability distribution. The transition function is denoted as $T : S \times A \times S \to [0, 1]$.

A reward function is used to provide a real number representing an incentive for being in a state, or taking a particular action when in a state. This leads to two forms of description: either $R : S \to \mathbb{R}$ or $R : S \times A \to \mathbb{R}$. The difference is whether the choice of action should be rewarded or not. When using probabilistic transitions, note that choosing action a does not guarantee a transition to s'. In this case, the reward is based on the expected outcome of the action. Developing a reward function can be a challenge. Russell and Norvig suggest ordering preferences and assigned ordered weights to the preferences [20].

Given the above descriptions, a MDP can then defined as a tuple $< S, A, T, R >$ where S is a finite set of states, A is a finite set of actions, T is a transition function and R is a reward function.

Choosing which action to take when in a particular state is referred to as a policy. A stochastic policy is given as $\pi : S \times A \to [0, 1]$. This means that actions which map to higher probabilities have greater preference. The lower probability choices may still have value for future learning if they have not been sufficiently explored. Applying a policy follows the sequence: initial state, assess policy to choose an action, receive the reward for the action, apply the transition for the action, transition to the new state, and so forth.

The horizon is the amount of time, or number of actions that an agent can take. Generally there are three kinds of horizons: fixed finite, indefinite finite and infinite. Russell and Norvig provide an excellent example of how the horizon length affects the value of following a policy [20]. In this example, an agent is attempting to navigate a maze. With a short horizon (fixed finite), it must target the maze exit as aggressively as possible. A longer horizon may allow an agent to explore the environment further to make safer options.

While rewards are used for individual state transitions, assessing the value of a policy requires a state history to be calculated. This is called a utility function, and is given as $U_h([s_1, s_2, \ldots, s_n])$. The two main ways to calculate this are with additive rewards or discounted rewards. Using discounted rewards, where γ is a discount factor bounded between $[0, 1)$, even infinite sequences can have a utility assigned. In general, the utility function is given as $U_h([s_1, s_2, \ldots, s_n]) = R(s_1) + \gamma R(s_2) + \gamma^2 R(s_3) + \ldots + \gamma^{n-1} R(s_n)$ However, in a stochastic environment the state transitions are not guaranteed, so policies are compared by using expected utilities. The expected utility of executing a policy π starting in state s is given as $U^{\pi}(s) = E[\sum_{t=0}^{\infty} \gamma^t R(S_t)]$.

Partially Observable MDPs

The initial MDP formalism assumed a fully observable environment that does not apply to many systems. Typically, noisy or limited sensors mean that agents cannot plan with knowledge of the full environment. Accounting for this begins by including a sensor model $P(e|s)$ where $e \epsilon E$ is the evidence received for the state s. E is the set of all potential evidence. A single evidence variable may be mapped to multiple states (limited sensors), or a single state may map to multiple evidence variables (noisy sensors). Evidence is often referred to as percepts, as in Section 2.2.3. These partially observable MDPs are referred to as POMDPs.

The key feature to deal with this uncertainty is a belief state, which represents the possible underlying states given the sequence of all actions and received percepts [20]. Note that currently the belief model is non-Markovian, as it relies on an indefinite sequence of actions and evidence. When an agent applies action a on a belief state b, it transitions to a new belief state b'. b' represents all possible states that the underlying states of b could have transitioned to given action a. When new evidence is received, the agent can filter out the underlying states that are not possible for b' to get a better representation of the true state distribution. This process mitigates the importance of an indefinite history of the action and evidence sequences and effectively converts a POMDP into a MDP on the belief space of the agent by making the belief space distribution dependent on only the evidence received from the previous state.

Spaan provides an overview of POMDPs in [25]. Solving these systems is extremely difficult computationally, so much of the research is focused on exploiting system properties to develop techniques such as factoring the state space or randomization strategies to select better learning samples. Monte Carlo Tree Search [24] and scalable planning [2] are a couple of such examples. Russell and Norvig suggest dynamic decision networks as a computationally efficient method to solve POMDPs [20]. This will be examined later in Section 2.3.5.

Bug Battle is a POMDP. Although abstracted as a MDP the outcome of actions is largely deterministic, but there are some circumstances where the outcome may vary. The element of partial observation is due to the limited sensor range and ability of cloaked bugs to remain undetected.

Solving a MDP

The goal of reinforcement learning is typically to find an optimal policy. MDPs have two general approaches to solve, value iteration or policy iteration. Each approach calculates the optimal policy for each state as an exact solution. When state spaces become larger, the sheer volume of calculations required makes these exact solutions intractable. For this reason most of the literature on MDPs or extensions of MDPs focus on approximate solutions. Several of the developed algorithms focus on the idea that only a very small portion of the state space needs to be considered in most cases. A brief overview of value iteration and policy iteration are given here for completeness. Full details are available in reinforcement learning texts such as [30].

Value iteration is based on the idea of calculating the utility for each state and then using the state utilities to select the best action to take [20]. It forms a recursive relationship called the Bellman equation. A system with n states has n associated Bellman equations. The Bellman equation uses the max operator, which makes the relationship non-linear and requiring an iterative solution.

Policy iteration seeks to improve on value iteration by interweaving two steps: policy evaluation and policy improvement. Policy evaluation is where the utility of each state is calculated if a given policy were followed. Policy improvement is where a new maximum expected utility policy is calculated using one-step look-ahead [20].

2.3.3 Algorithm Review

There are several approaches to solving RL systems throughout the literature. This section briefly covers a few of the basic algorithms, although it does not go into extensive detail. Most of these algorithms are not appropriate for Bug Battle because they are based on the assumption of a static environment.

Temporal-Difference Learning

Temporal-difference learning (TD learning) has a utility function and uses the differences of temporally successive states to adjust utilities of observed states while a simulation is running [27, 26]. This allows incremental updates referred to as bootstrapping. Note that actions chosen are not explicitly considered in this approach.

Q-Learning

In Section 2.3.2 the utility function was presented, which considers values of transitions between states. The actions were not considered directly in this function. If the utility function is extended to account for actions, we get the utility-action function $Q : S \times A \to [0, 1]$, which is often just referred to as the Q-function [31, 30].

Q-learning is based on using the Q-function, which attempts to find the maximum expected utility of taking an action in a given state [20]. The main advantage of Q-learning over TD learning is that is does not need the transition function which means it is a model-free algorithm. Further, Q-learning uses the best Q-value, which means it is learning independently of the actual policy being followed. This is called off-policy learning.

State-Action-Reward-State-Action (SARSA)

SARSA is very similar to Q-learning, except it is an on-policy algorithm [20]. The learning is applied from the actions that have actually been performed, not just the best action(s).

Monte Carlo Methods

The basic idea behind a Monte Carlo method is to evaluate randomized input to develop an estimate for a stochastic process [13]. Monte Carlo Tree Search is an extension that attempts to estimate the outcome of a simulation by calculating the average outcome of simulations from a given state. It has been extended to POMDPs such as Partially Observable Monte-Carlo Planning (POMCP) [24], but it is not being considered for Bug Battle because the approach predicated on Bayesian networks was assessed to be more practical to implement within the given time constraints.

2.3.4 Hierarchical Task Networks

The curse of dimensionality was coined by Bellman in the 1950s [3]. It means that problems will be difficult to solve with some algorithms because the size of the problem space is just too large. Attempting to solve this curse has led to a variety of innovations such as heuristics (functions to guide algorithms to approximate solutions quickly), sampling methods such as Monte Carlo or abstracting the problem to reduce the overall size. Hierarchical Task Network (HTN) are based on the principle of abstraction. The core definitions for HTNs come from Russell and Norvig unless otherwise stated [20].

HTNs use hierarchical decomposition to separate a problem into component parts to be shared and reused. The main benefit is that at any level of abstraction the number of activities to plan is much smaller, which can be solved more easily than a large problem of individual components [20]. Another benefit to abstracting activities is that preconditions are easier to manage computationally and intellectually.

HTNs divide actions into two sets: primitive actions or High-Level Actions (HLAs). HLAs may sometimes be referred to as macros or abstract actions. HLAs contain one or more refinements, which are either sequences of other HLAs, primitive actions or some combination of the two. An HLA that consists of only primitive actions is called an implementation of the HLA.

2.3.5 Bayesian Networks and Extensions

This section develops the concept of DDN beginning with the basic Bayesian Network (BN) and adding features as complications are considered. While not examined in this review, HTNs can be combined with DDNs to yield Hierarchical Dynamic Decision Networks (HDDNs) [29]. HDDNs are a natural extension that offer dramatic speed increases to consider for future work. They were considered to be out of scope for this thesis.

Bayesian Network

BNs are a data structure to concisely represent dependencies amongst variables [20][9]. A BN can be represented as a directed graph where each node is a random variable (r.v.) and each edge indicates dependence. For an edge connecting node X to node Y, X would be called the parent of Y. For every node X, $P(X|Parents(X))$ is the conditional probability distribution.

Bayesian networks can be thought of as belief networks because the underlying probabilities for the nodes represent the degree of belief that the event the node represents can occur. Representing a belief network this way simplifies the task required for POMDPs (see Section 2.3.2). Creating the network with expert domain knowledge can simplify the learning process, but it risks obscuring actual relationships if built incorrectly. Algorithms exist to both build the structure of a network and populate the probabilities [11]. Their implementation was considered out of scope for this research.

[9] A model can be represented as a joint probability table, but the resulting representation is extremely large. As an example, a system with just 10 variables would require $2^{10} = 1024$ entries to describe the system. This does not scale well, as doubling the quantity of variables then requires $2^{20} = 1{,}048{,}576$ entries.

A system with completely dependent variables would result in a fully-connected graph which is no better than the joint probability distribution, but this is often not the case. Sparse networks will have few edges, which makes them much easier to solve computationally and understand intellectually.

Dynamic Bayesian Network

Moving from a static system (each r.v. is fixed) to a dynamic one (r.v.s can change over time) causes a problem for standard BNs. To capture the temporal aspect, a dynamic system is divided into a series of time slices, where the BN is effectively replicated across time slices [20]. The variables that are hidden or observable could vary by time, but it is easier to consider them fixed. This changes the notation of state S and evidence E to S_t and E_t to account for the time-dependence.

The concepts of this section are reviewed in several textbooks on artificial intelligence or machine learning, [20, 1, 17, 11] all have similar presentations of the material.

Since the system can change, as in the case of an MDP, a transition model is required. This model takes the same form $P(S_t|S_{t-1})$, which includes the Markov assumption that the current state only depends on a (fixed number of) previous state(s). In general, the dependence could be longer than one state but one is assumed to be sufficient for analysis of Bug Battle.

Similarly, making the Markov assumption for the sensor moves the sensor model from a general case to a more tractable one. In this case, the current state generates the current sensor readings which means there is no history dependence on previous states. The sensor (or observation) model is given as $P(E_t|S_t)$.

A potential problem is that the distributions between variables could change between time slices. This is often assumed away by adopting the idea of stationarity, which keeps the distributions (not the variables) the same across time slices. It simplifies the model considerably but does not always apply. [14] describes some work where non-stationarity could be modelled. Bug Battle could be view as either a stationary or non-stationary process, depending on the state representation. To keep the thesis within scope, emphasis is placed on modelling the problem as a stationary process.

Introducing time to the BN representation allows time-based inferences. The state can be estimated at the current time (filtering), past (smoothing) or future (prediction). While the purposes of filtering and prediction are readily apparent, smoothing may be less intuitive since there was already a state estimation at that past state. Smoothing uses historical and current

observations to obtain better estimates of previous state distributions [17, 20]. It can effectively be thought of as backwards propagation - if new evidence makes some elements of a prior distribution impossible, the estimate for the remaining distribution is improved which aids future prediction.

Particle filtering is a particular approximation algorithm that can be used for filtering [17, 20, 14, 8]. The general idea is to sample the current distribution on the belief space, weight for the evidence and then re-sample from weighted evidence to obtain an unweighted sample. This has the effect of reinforcing samples that have a high probability and culling samples that have a low probability. This method is especially useful when the full probability model is not know in advance. A detailed example of applying particle filtering is provided in Section C.1.

Decision Network

The purpose of reinforcement learning is to have agents act within an environment to learn how to make better decisions in the future. BNs and Dynamic Bayesian Networks (DBNs) are lacking the element of decisions. Once decisions are permitted, there needs to be some way to indicate which decisions are better than others. This can begin by stating preferences for actions in a situation. With some constraints these ordered preferences can be developed into an underlying utility function [20, 17]. Rational agents are expected to choose the highest expected utility, which is called the principle of maximum expected utility.

Decision Networks (DNs) add decision nodes and utility nodes to BNs. Decision nodes are similar to r.v.s but represent points where an agent has a choice among a set of actions. Utility nodes represent the utility function. The parents of the utility node are any variables or decision that directly affect the utility.

Solving a decision network is similar to solving a BN, but each possible value of the decision node must be considered. These decisions are then pushed forward in the network to calculate the utility for each action, and the action that results in the highest utility should be chosen [20]. The presence of a large set of actions will significantly increase the computation time for this algorithm.

Dynamic Decision Network

DDNs combine DBNs with DNs (note that DNs extended BNs and not DBNs). Thus, a DDN can be thought of as a network spanning multiple time slices

that incorporates agent decisions and utilities. Filtering algorithms update the belief state representation and decisions are made by projecting forward the possible action sequences to choose the best one [20].

There are three general methods to solving a DDN: variable elimination, value iteration and solving a decision tree. Turkett and Rose suggest that variable elimination is the best of these methods as it capitalizes on the structure of the BN [29]. However, variable elimination assumes full knowledge of system probabilities, which is not always the case. For this reason solving a decision tree is chosen. A decision tree is a branching structure that considers the current state and builds branches for each possible decision while tracking a overall reward structure in the nodes of the tree [20]. An example of building and solving a decision tree is given in Section C.2. Decision trees can be built with approximate methods and are useful when exact solutions are intractable.

While the time complexity for making decisions in a DDN is far better than using value iteration to solve a POMDP, there are still potential improvements [20][10]. One such improvement is HDDNs [29], as mentioned at the start of this Section, although it is out of scope[11].

2.4 Summary

ABMs are low-level abstractions of a system that model the mental and physical states of an actor within an environment. One benefit of dissecting ABM behaviour is to gain a deeper understanding of the emergent characteristics of a system to guide future development.

While computationally expensive, machine learning is a useful technique to explore a system when the scope of interaction is high. Bayesian networks and their variants (DBNs, HTNs, DDNs, HDDNs) capitalize on underlying conditional probability structures to reduce the computational cost. "Learning" is achieved in a DDN through the filtering process, which revises an estimate of the true state based on observations. Future decisions are then guided by maximizing rewards to optimize agent behaviour.

[10] The time complexity for DDNs is $O(|A|^d|E|^d)$ while the time complexity for value iteration is $|A|^{O(|E|^{d-1})}$. In both these cases, $|A|$ is the cardinality of the action set, $|E|$ is the cardinality of the evidence set, and d is the number of decisions projected forward.

[11] The HDDNs examined in [29] were based on established problems where DDN constructs already existed. The decision to keep HDDNs out of scope was based on lack of experience with DDNs and the scheduling risk to develop both was deemed too large.

3 Model Design

3.1 Introduction

In order to test the hypothesis that the simulation can be treated as a POMDP solvable through a DDN, it is necessary to have an adversary to compare the Dynamic agent against. The strongest candidate from previous years' submissions to the Bug Battle contest is the expert bug "Rathbug", whose development began in the original BugWars. Due to framework changes between BugWars and Bug Battle, only the multiple spawning (Section B.3.4) and energy transfer (Section B.3.5) techniques have been credited in prior work.

A thorough review of the Bug Battle simulation is provided in Annex B. This chapter expands on the concepts of that review to explain the designs of the agents involved in the thesis testing. The chapter is divided into the following major sections:

- Presentation of physical designs available to both the Baseline and Dynamic agents;
- Explanation of the behavioural design of the Baseline agent; and
- Explanation of the behavioural design of the Dynamic agent, which includes a description of the implementation and solving of the DDN.

3.2 Common Agent Variants

This research assumes that all agents in the experiments will have access to the same conceptual physical models. It is possible that better organ build variants exist[1]. Exploration of the physical model of bugs was not performed

[1]For example, additional variants were developed in the initial design but discarded due to complications in use. One such variant was a "Soldier" bug, which was permanently cloaked with an energy sensor, budder and seven cilia (movement organs). The concept of the soldier bug was to be spawned onto the edge of an opposing colony where it would begin undetected. On its turn the soldier bug would penetrate deeply into the colony, harvesting the energy from the exposed colony interior. The budder would allow it to grow like a

due to scope constraints.

3.2.1 Physical Model

The sections below describe the build order of organs and the rationale behind the design of each variant. The rationale descriptions are based on the employment by the Baseline agent. The Dynamic agent uses the same core physical design, but chooses which action to take based on the DDN selection. In the organ build tables, there are two times that organs are added: either the initial build when the bug is created or during the bugs turn as extra energy is available.

Scout

The Scout variant is used to rapidly explore the environment with a focus on exploiting plants and weaker enemies. Stronger enemies are avoided. This variant retains minimal energy from turn-to-turn by focusing on spawning as many children as possible. The goals of this design are to cover the map as fast as possible, and to aggressively target weaker enemies. The minimum energy is based on the probability of finding energy sources on the next turn. Note that plants regenerate, so while individual bugs will die off quickly, a Scout colony as a whole can achieve a stable population[2].

The Scout variant build order is given as Table 3.1. Note that all organs are part of the initial build. This is based on the assumption that having sufficient energy for maximum expansion is ideal. The alternative would be to start with a reduced quantity of cilia, but it would require considerable parameter-tuning to be effective.

Worker

This variant is designed as the workhorse of the agent colony. Its goal is to generate energy while remaining adaptive to the environment. A major anomaly in the Worker bug relative to most designs is that it is immobile. The variant uses active and passive defense to protect the colony. The final poison capability allows the bug to spit poison on adjacent adversaries, or to inflict damage if the adversary accidentally initiates combat. Further passive defense

cancer within. However, since it is rare for dense colonies to form when employing the rapid movement of the Scout variants, the potential usefulness of the Soldier was deemed too low to implement.

[2]For discussion, a colony is defined as "a collection of bugs which may or may not be adjacent to each other".

Table 3.1: Scout Organ Build

Order	Organ	Added
1	EnergySensor	Initial
2	Budder	Initial
3	Cilia	Initial
4	Cilia	Initial
5	Cilia	Initial
6	Cilia	Initial
7	Cilia	Initial
8	Cilia	Initial
9	Cilia	Initial
10	Cilia	Initial

Table 3.2: Worker Organ Build

Order	Organ	Added
1	Cloaking	Initial
2	PhotoGland	Initial
3	PhotoGland	Initial
4	PhotoGland	Initial
5	PhotoGland	Evolve
6	EnergySensor	Evolve
7	Budder	Evolve
8	PhotoGland	Evolve
9	PhotoGland	Evolve
10	PoisonGland	Evolve

is achieved by always remaining cloaked. Further active defense is achieved via adaptive spawning to create scenario-specific variants, which include all variants presented in this work.

While all variants are capable of the energy transfer mechanism (see Section B.3.5), it is primarily intended for the Worker variant. Typically the Worker will be in a clustered colony of other Worker bugs. Spawning inside the colony uses the energy transfer mechanism. Spawning on the colony fringe follows several potential paths depending on detection of enemies. If enemies are detected, then they should be cleared using attacker bugs.

In the case that cloaking is suspected, this variant is capable of employing the techniques described in Section B.3.9 to target potential cloaked bugs.

Attacker

The attacker bug is designed to be spawned directly on an enemy, and die immediately. The organ build is intended to maximize defensive damage based on the amount of energy supplied to the bug. While this build will often be one or more poison glands, in some cases it could be spikes. More detail on how the organ build is selected is given in Section B.3.10.

Feeder

This variant exists purely to transfer energy from a source bug to a target bug. It has no organs, as adding organs would waste energy. Feeders are designed to die as soon as they are born, which transfers all available energy to the recipient bug.

Recce

This variant exists purely to detect if another bug is in a cell to support the recce technique (see Section B.3.9). It has no organs, as adding organs would waste energy. Although the organ build is the same as the Feeder variant, the conceptual employment is completely different so a different name is used to distinguish the purpose.

3.3 Baseline Agent Design

The Baseline design prioritizes expansion at the expense of survivability initially, and then transfers to probabilistic consolidation. In the expansion phase, it spawns aggressively to exploit resources (using the Scout variant, see Section 3.2.1), which will fuel further growth. In the consolidation phase, it uses the Worker variant (see Section 3.2.1) to convert excess energy to the base of a colony.

The transition from expansion to consolidation is based on the number of turns to cover the map in an unopposed environment, bug energy levels, and a random variable relative to Scout population and Worker population[3]. The probability function used to calculate the likelihood of spawning a Worker bug is given as Figure 3.1.

[3] This function was created and calibrated during preliminary testing in an unopposed environment. Calibration was based on face validation to ensure Worker variants were being spawn early yet not exclusively.

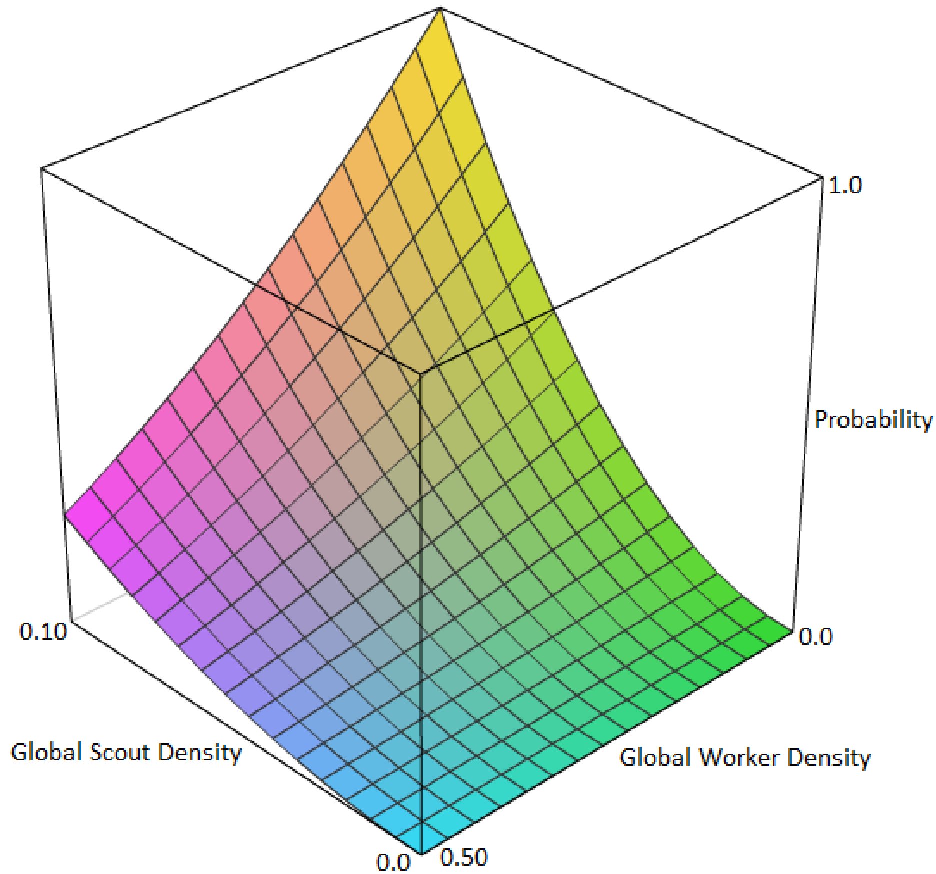


Figure 3.1: Probability of Spawning Baseline Worker Agents

The purpose of the probabilistic component is to ensure that an active mobile defense is maintained when the transition begins. Otherwise all Scouts may attempt to convert to Workers immediately, which would allow adversaries a prolonged respite while Workers develop their internal organs.

By placing all effort towards exploiting resources this design is able to cover the entire map in approximately ten turns[4]. While the Scout bugs may not have visited every cell, it will have attacked the vast majority of resources (plant or adversary) encountered.

[4]Found by preliminary testing by placing the Baseline variant in an unopposed environment and allowing the simulation to run until the initial plant resources were depleted. The coverage pattern that the bug exhibits when expanding makes the outer edge of the dispersed colony obvious.

Table 3.3: Baseline Movement Priorities

Priority	Description
1	(Strongest of) weaker enemies
2	(Strongest of non-isolated) plants
3	(Strongest of isolated) plants
4	Default direction (if not already occupied by an ally)
5	Open ground

Table 3.4: Baseline Spawning Priorities - Scout

Priority	Description
1	(Strongest) weaker enemy
2	(Strongest) plant
3	Open ground

3.3.1 Behavioural Model

The evolution of the Baseline variant always uses as much energy as available to complete the physical build while retaining sufficient energy to sense the environment and act upon these senses. The strategy here is to move as much as possible to find resources, which will allow additional movement and spawning.

The design uses the frequency-based exploration technique to choose directions (see Section B.3.2). After it senses the local environment, it prioritizes the search results as given in Table 3.3. In this table, a "non-isolated plant" means a plant that has adjacent plant neighbours.

Note that stronger enemies are absent. Like the tactic of blitzkrieg, strong pockets of resistance are ignored in favour of destroying their support. Often when encountering a stronger enemy, the Baseline variant would have sufficient energy to use the kamikaze technique (see Section B.3.6) to weaken it, but this is energy that can fuel further exploration and degrade the long-term survivability of the (temporarily) stronger adversary.

When spawning, the design prioritizes as given in Table 3.4. The strategy here is similar to the movement plan. Note that the default direction assigned to the newly spawned bug is orthogonal to the parent bug, which encourages faster dispersion across the world. On each round of movement the parent is limited to spawning three children so that it retains sufficient energy and open space to explore.

Once the transition is triggered, spawning of Worker variants may begin.

Table 3.5: Baseline Spawning Priorities - Worker

Priority	Target	Variant Spawned
1	(Strongest) enemy (if poison available)	Poison target, then spawn Scout
2	(Strongest) plant	Scout
3	(Strongest) weaker enemy	Worker
4	Open ground	Worker

Workers are spawned with enough energy to be self-sufficient. If they are spawned with just enough energy for their initial evolution it will take them several turns to develop moderate energy generation. However, when the Scout bugs delay spawning until the end of the turn and collect several food sources, they will spawn a Worker approaching maximum evolution.

Recalling that Workers are immobile means that they have no movement decisions to make. Their spawning decisions are more complex and are given in Table 3.5. The purpose of this design was to ensure that there are as few plants adjacent to the colony as possible. Since the Workers are cloaked, adversary bugs cannot find them directly, which means they cannot target them. However, adversaries could target the plants which would place the adversary adjacent to a Worker. Therefore spawning Scout variants on plants denies these resources to the adversary and provides an active defensive mechanism to destroy the adversary if they are weaker.

At the end of a spawning phase, the Worker variant will spit poison onto adjacent enemies, prioritizing stronger enemies. This is only effective on the first spawning phase, because the poison reservoir may become depleted and will not be refilled until the end of the turn. Since the Worker typically uses all available surplus energy for spawning, this is only a minor concern.

3.4 Dynamic Agent Design

Dynamic bugs execute their turns according to the algorithm presented in 3.2. Evolution is a fixed process to ensure that sensor organs are available to scan the local environment[5]. A Dynamic bug then uses the percepts and probabilities to make a decision according to the DDN, solved using a decision tree which is built using reward and utility functions. The result of the DDN

[5]Recall from Annex B that bugs can be spawned on an enemy to trigger combat, and they may delay adding organs so that they have additional energy to win combat. The decision-action cycle is the first time that organs can be added post-combat.

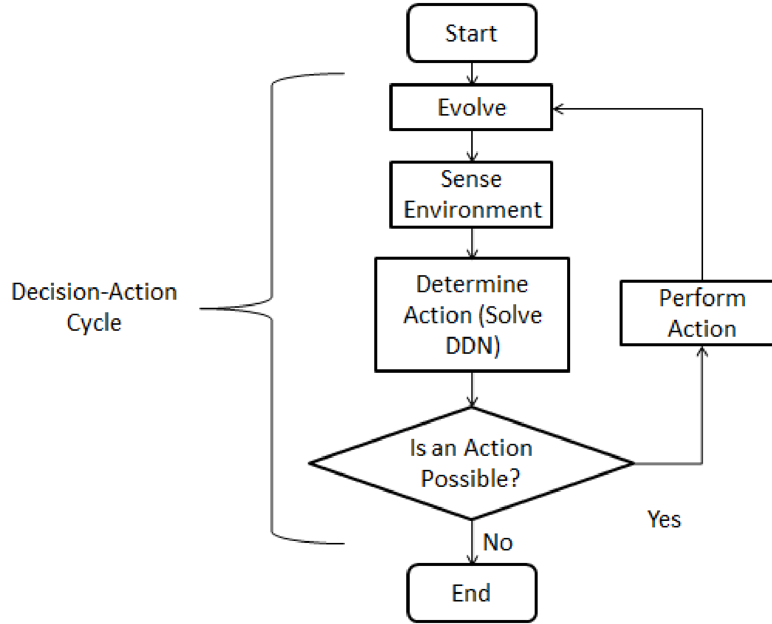


Figure 3.2: Decision-Action Cycle for a Dynamic Bug Turn

is used to execute a low-level action. This decision-action cycle continues until the bug cannot execute additional actions.

3.4.1 Reward and Utility Functions

Development of reward and utility functions are a critical component of a DDN, similar to how development of a fitness function is fundamental to a genetic algorithm [34]. At the highest level the reward would be success or failure of the Dynamic bug in eliminating the Baseline bug. At a lower level, the only "reward" the environment offers is energy.

The reward function was defined as addition of weighted sub-functions. This representation facilitated independent manipulation of sub-function weights. The sub-functions were designed based on the same heuristics that drove the implementation of the Baseline bug. The reward function is:

$$\begin{aligned} R(S, A, S') = {} & w_1 f_1(S, A) + w_2 f_2(S, A, S') + w_3 f_3(A) \\ & + w_4 f_4(A) + w_5 f_5(S) \end{aligned} \tag{3.1}$$

where:

- S is the initial state

- A is the action taken
- S' is the transitioned state
- $w_i \epsilon [0, 2]$, $i = \{1, 2, \ldots, 5\}$
- $f_1(S, A)$ rewards the energy
- $f_2(S, A, S')$ rewards exploration
- $f_3(A)$ rewards killing enemy bugs
- $f_4(A)$ rewards spawning new bugs
- $f_5(S)$ rewards the defensive posture of the bug

Similarly, the utility function considers the average energy and defensive rating of all samples in a belief node (see Section 3.4.3 for a discussion of belief nodes).

The implementation of the sub-functions is fixed and based on scaling the desirability of the intent of the factor based on heuristics. For example, $f_2(S, A, S')$ models a bonus for exploring the environment based on the search-scoring algorithm described in Section B.3.2[6]. An overview of the variable sub-functions is provided in Figure 3.3[7]. The enemy and spawning sub-functions were fixed rewards dependent on the action chosen.

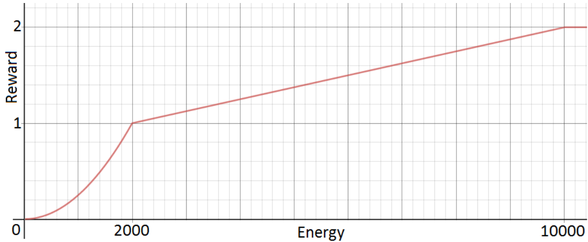
Factor weights are defined on a continuous scale relative to the accuracy of the expert analysis. A weight of zero would imply the expert opinion of the factor was not relevant. A weight of one would imply that the factor was assessed accurately, and a weight of two would imply that the factor was significantly more important than it was assumed. The weights of the reward function are fixed for a given trial. Varying the weights during a trial is possible but it was not explored.

3.4.2 DDN Implementation

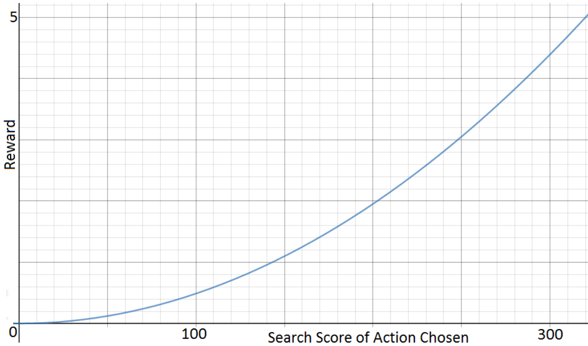
The additional probabilities that are included in the DDN planner come from a central controller (see Section B.3.7). When Dynamic bugs act within the environment, some statistics are generated for recently observed effects. Regionally-dependent statistics allow the DDN planner to sample the unknown environment based on recent history, which allow the Dynamic bugs

[6] Initially this should be crucial for expansion and dispersion of the bugs, to capture as many of the plant resources as possible. As the local region the bug explores is further explored, the relative importance of this factor diminishes.

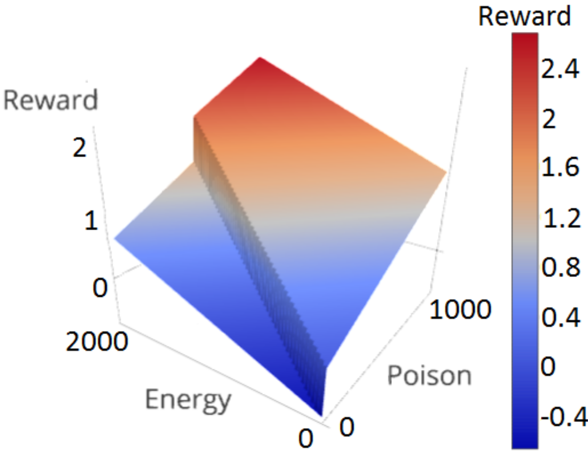
[7] These were the intended relationships. However, when reviewing the simulation code to produce the graphs it was discovered that there was a mistake between min and max operators for the energy and defensive sub-functions. The implemented energy sub-function does not have the smooth transition at $energy = 2000$. The defensive sub-function has a much larger upper range (approximately 8 instead of 2). The impact of these errors is unknown. This is discussed further in Section 6.2.1



(a) Energy Sub-Function



(b) Exploration Sub-Function



(c) Defense Sub-Function

Figure 3.3: Reward Sub-Functions

Table 3.6: DDN Focus

Description	Concept
AGGRESSIVE	Attempt to damage or kill enemy
DEFENSIVE	Make the bug more difficult to kill
DISPERSIVE	Spread out across the map
ENERGIZE	Prioritize gathering energy

to adapt to the environment as the simulation progresses. For example, initially the probability of encountering enemy bugs is almost zero. If several enemies are encountered in a short time period, the DDN planner assumes a higher chance of additional enemies when exploring the local environment. If no enemies are then encountered for a few simulation turns, the DDN planner assumes the enemies have passed through the region and the probabilities are adjusted.

The DDN planner considers four core foci (see Table 3.6), which define primary and secondary intent for a strategy. By considering each combination of foci, there are 16 possible strategies. Each strategy maps to zero or more low-level actions (see Table 3.7), which have a tactically optimal implementation once chosen. When a strategy is chosen by the DDN planner, the actual action to implement is chosen at random. However, the actions are filtered before executing so there is typically only one action left for a given strategy.

The decision to structure the DDN planner in this strategy-focus-action construct was partially based on the work of HDDNs (see Section 2.3.5) and partially to provide a tractable abstraction to implement. Attempting to determine the combination of factors (e.g. direction, amount of energy to use, etc) for every possible organ proved cumbersome in preliminary designs. Using effect-based notation vastly simplified implementation.

3.4.3 Solving the DDN

Solving the DDN amounts to solving a unique decision tree. This must be done for each decision-action cycle. Nodes in the decision tree correspond to belief nodes or chance nodes. Particle filtering (see Section 2.3.5) is used to filter the belief nodes when required. The figures for this discussion are based on the toy model used for preliminary experiments (see Section 4.2.1) to keep the scope of the figures sufficiently small. The process is the same for the Bug Battle decision tree, but the scope is much larger due to the cardinality of the action set.

Table 3.7: DDN Action

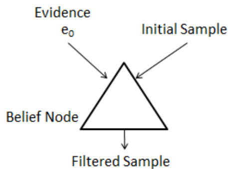
Primary Focus	Secondary Focus	Action Name
AGGRESSIVE	AGGRESSIVE	SPAWN-ATTACKER-TO-KILL
AGGRESSIVE	DEFENSIVE	SPIT-POISON-TO-WEAKEN-FOR-WORKER SPAWN-ATTACKER-TO-WEAKEN-FOR-WORKER SPAWN-WORKER-ENEMY
AGGRESSIVE	DISPERSIVE	SPIT-POISON-TO-WEAKEN-FOR-SCOUT SPAWN-ATTACKER-TO-WEAKEN-FOR-SCOUT SPAWN-SCOUT-ENEMY
AGGRESSIVE	ENERGIZE	-
DEFENSIVE	AGGRESSIVE	SPIT-POISON-TO-KILL
DEFENSIVE	DEFENSIVE	FILL-POISON
DEFENSIVE	DISPERSIVE	SPAWN-RECCE
DEFENSIVE	ENERGIZE	TRANSFER-ENERGY
DISPERSIVE	AGGRESSIVE	-
DISPERSIVE	DEFENSIVE	SPAWN-WORKER-BEST-SEARCH
DISPERSIVE	DISPERSIVE	SPAWN-SCOUT-BEST-SEARCH
DISPERSIVE	ENERGIZE	MOVE-TO-BEST-SEARCH
ENERGIZE	AGGRESSIVE	MOVE-TO-ENEMY
ENERGIZE	DEFENSIVE	SPAWN-WORKER-BEST-ENERGY
ENERGIZE	DISPERSIVE	SPAWN-SCOUT-BEST-ENERGY
ENERGIZE	ENERGIZE	MOVE-TO-BEST-ENERGY

A belief node is a combination of environment percepts and internal bug state. The environment percepts are based on locations, energy signatures and underlying probabilities. The type of bug present at each cell is based on a combination of energy signature and probability - for example, while an energy reading of 950 is most likely to be a plant, there is a small chance it could be an enemy. Some probabilities are known with certainty based on the tracking functions of the centralized “HQ”.

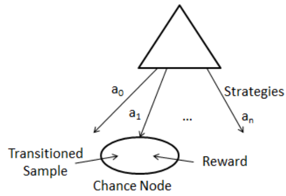
The root belief state is generated based on the initial sensor input, and filtered for the current probabilities (see Figure 3.4a)[8]. The result is several samples of the belief space. The planner then begins to consider every possible strategy, but filters out strategies that are impossible. Strategies removed due to filtering may be due to underlying action(s) lacking energy, pre-requisite organs, or suitable targets in the environment.

A chance node is obtained by having each remaining strategy obtain a copy of the samples for the parent belief node and transition each sample according to the action(s) mapped to the strategy. New environment information is

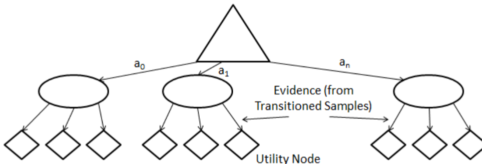
[8]In the figure it is assumed that the start state is the prior distribution of the toy model. Further, to be consistent with the literature such as [20, 17], triangles are belief nodes, ovals are chance nodes and diamonds are utility nodes.



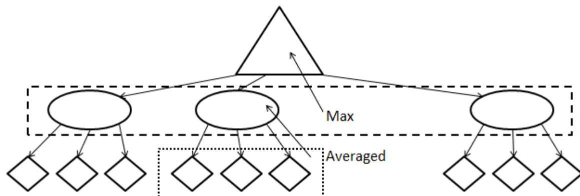
(a) Root Belief Node



(b) Creating Chance Nodes



(c) Full Decision Tree



(d) Solving the Tree

Figure 3.4: Building and Solving a Decision Tree - 1-step

based on sampling the system probabilities. Rewards are applied based on the relative quantity of samples in each transitioned state and the action chosen (Figure 3.4b). Since the decision tree is forward planning, the rewards are being calculated from the belief state space not the true state space.

Before a chance node propagates to additional belief nodes, it generates and weighs all evidence for transitioned samples. A new population is then sampled for the weighted evidence. This effectively prunes branches of the tree that are very unlikely. The link from a chance node to a child belief node is the evidence generated. Figure 3.4c provides an example. When

the maximum depth of the tree is reached, each leaf belief node is evaluated against a heuristic utility function.

The reward function is a linear combination of factors based on expert analysis of the simulation, to provide a short-term value for each belief state. Similarly, the utility function provides an estimate of the long-term value of each belief state.

Once built, the decision tree is easily solved: belief nodes obtain their final score from the maximum of their child chance nodes. Chance nodes obtain their final score from the average of their belief nodes. The end effect is the root belief node selecting the strategy that led to the highest score. Ties between child nodes can be resolved randomly. Figure 3.4d provides a summary of this process.

3.5 Summary

While both the Baseline and Dynamic bugs use the same physical designs, the Baseline bug uses fixed logic based on hard-coded conditions determined by expert analysis of the simulation. The Dynamic bug makes very few firm decisions, instead relying on a decision-action cycle fed by sensor input and probabilities obtained from a central controller.

The key feature the Dynamic bug relies on is solving a DDN by building and solving a decision tree constructed with particle filtering. A major challenge with building the decision tree is the selection of the reward and utility functions. Continually building and solving decision trees based on the current environment and observations allows flexibility and adaptability. However, the quantity of calculations required makes this approach very computationally expensive.

4 Experiment Design

4.1 Introduction

This chapter is divided into the following major sections:

- Parameter selection experiment design; and
- Distributed environment.

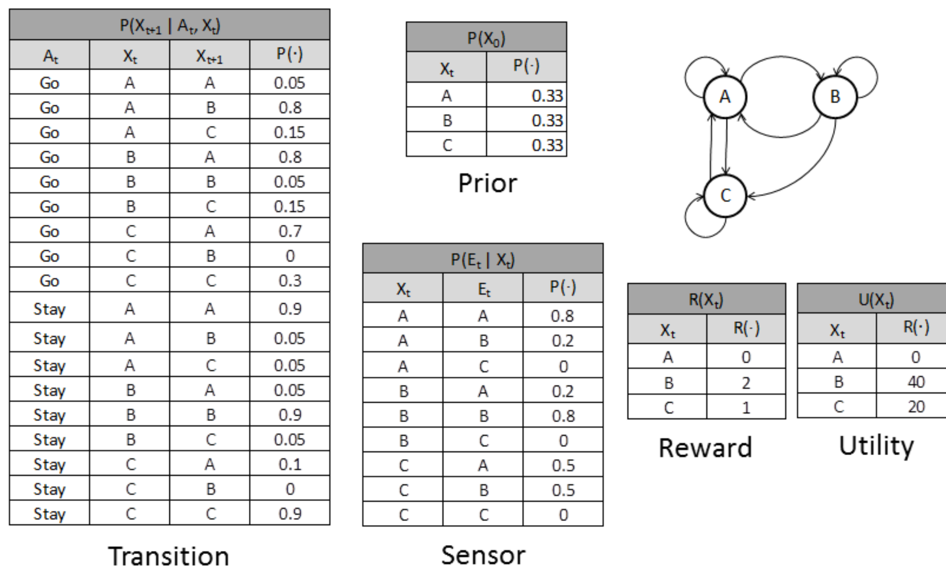
4.2 Parameter Selection Experiment Design

Since learning in a DDN is determined by the filtering of belief spaces, testing simulation performance meant either varying the particle filter population size and decision tree search depth or the reward and utility functions. While a combination of these approaches could be used, the literature review emphasized the high computational cost of DDN solving, so a preliminary experiment was designed to estimate the cost and benefits of particle filtering with the intention of guiding which approach to use.

4.2.1 Preliminary Experiment - A Simplified Model

The toy system in the preliminary experiment was a 3-state model as given in Figure 4.1 developed independently to test simplified characteristics present within a POMDP. This system captured stochastic transitions, partial observation (noisy sensors) and a hidden state. The probabilities for the various sub-models (prior, transition, sensor, reward and utility) were arbitrarily chosen.

The preliminary experiment also allowed verification of a generic DDN algorithm implementation which could later be adapted the full experimental model. In order to be considered a valid verification model, the features tested in the toy model had to map to features in the larger-scale model. Stochastic transitions are relevant because the outcome of some actions in Bug Battle



$P(X_{t+1} \mid A_t, X_t)$			
A_t	X_t	X_{t+1}	P(·)
Go	A	A	0.05
Go	A	B	0.8
Go	A	C	0.15
Go	B	A	0.8
Go	B	B	0.05
Go	B	C	0.15
Go	C	A	0.7
Go	C	B	0
Go	C	C	0.3
Stay	A	A	0.9
Stay	A	B	0.05
Stay	A	C	0.05
Stay	B	A	0.05
Stay	B	B	0.9
Stay	B	C	0.05
Stay	C	A	0.1
Stay	C	B	0
Stay	C	C	0.9

Transition

$P(X_0)$	
X_t	P(·)
A	0.33
B	0.33
C	0.33

Prior

$P(E_t \mid X_t)$		
X_t	E_t	P(·)
A	A	0.8
A	B	0.2
A	C	0
B	A	0.2
B	B	0.8
B	C	0
C	A	0.5
C	B	0.5
C	C	0

Sensor

$R(X_t)$	
X_t	R(·)
A	0
B	2
C	1

Reward

$U(X_t)$	
X_t	R(·)
A	0
B	40
C	20

Utility

Figure 4.1: Toy 3-State Model

is unknown, as agents have no way to sense adversary cloaking or defensive damage with certainty[1]. The noisy sensors and hidden state replicate the effects caused by the probabilistic target detection (see Section B.2.4 and Section B.2.5).

Recalling Section 2.3.2, the prior table gives the probability of the state at system initialization. A percept is generated for an agent according to the sensor model and the agent can use this to choose one of two actions, $A = \{GO, STAY\}$. The sample is transitioned according to the action and initial state, and a reward is obtained depending on the true state of the system[2]. If an agent is employing a DDN, it uses the utility function to decide the "long-term" value of the state. For the purpose of this preliminary experiment, each agent had 200 turns to obtain the best score possible. There were 1000 trials for each utility function.

The toy model had a dominant strategy of attempting to reach state B and

[1]This is restricted to movement or spawning actions. Actions such as filling the poison gland are deterministic.

[2]Note that since the rewards are distinct in this scenario, it would be possibly to determine the true state with perfect accuracy. This could be mitigated by adding a small random component to the reward so that there was potential overlap between state rewards, or having the same reward for multiple states. This singular reward-to-state mapping seldom applies to full-scale models.

Table 4.1: Preliminary Experiment Results

Population	Depth	Time (ms)	Avg Reward	Utility Function
100	2	63.6	141.6	$P(X'\|STAY, X)^{10} * R(X)$
100	3	201.9	141.4	$P(X'\|STAY, X)^{10} * R(X)$
1000	2	398.7	141.5	$P(X'\|STAY, X)^{10} * R(X)$
1000	3	1601.7	141.7	$P(X'\|STAY, X)^{10} * R(X)$
100	2	65.2	139.2	Focus on B - 10
100	3	201.6	138.8	Focus on B - 10
1000	2	396.5	141.6	Focus on B - 10
1000	3	1601.1	142.1	Focus on B - 10
100	2	65.8	132.9	Focus on B - 100
100	3	201.1	131.2	Focus on B - 100
1000	2	402.7	141.0	Focus on B - 100
1000	3	1619.3	141.0	Focus on B - 100
...	...	...	...	...

then staying there indefinitely. Sensitivity testing was performed by varying the population size, search depth and utility function to determine the overall effect. In general most settings tended to under-perform the simple dominant strategy, largely due to a lag effect caused by sampled populations entering state C and then needing a turn or two to re-adjust to either state A or B as new evidence was received.

Some of the results from the preliminary experiment are presented in Table 4.1[3]. The average differences between population sizes are presented in Table 4.2. The values in Table 4.2 were calculated based on data from all utility functions. The significant differences in average reward are largely due to refinements in sampling in the particle filter. The small sample sizes are unlikely to generate a reasonably accurate prediction of the current state, which tends to lead to ineffective decisions.

While the effect of the utility function was alluded to in Section 2.3.2, it is clear from Table 4.1 that the design of the utility function has a effect on the overall results. The reward function was not varied in this model because it came directly from the environment. The trade-off of reward between population sizes of 100-1000 was relatively small given the significant time savings.

[3]The full table is almost 300 rows long. This table was edited to highlight the main relationships only. For comparison purposes, a planner that followed the dominant strategy scored an average reward of 141.7.

Table 4.2: Preliminary Experiment Summary

Population	Depth	Time (ms)	Avg Reward
10	2	23.8	120.9
10	3	61.1	119.2
100	2	65.0	139.9
100	3	203.2	139.4
1000	2	405.7	141.5
1000	3	1645.7	141.4

4.2.2 Parameter Selections

The results from the preliminary experiment gave justification to fix both population size and search depth and to vary reward and utility functions. Given the minor change in outcome relative to processing time, the fixed population size was chosen to be 100 samples and the maximum search depth was 2.

Recall from Section 3.4.1 that the reward function consisted of 5 factors while the utility function had 2 factors. Ideally factor weight parameter exploration could be executed first as a coarse parameter search followed by fine-tuning, but time analysis (see Section 4.3) indicated that this would not be feasible. Limited time led to the decision to explore weights of 0, 1 and 2 which is effectively only a very coarse parameter search.

4.3 Distributed Environment

The hardware requirements to run the experiment were a concern. Preliminary trials indicated that a typical run-time was between 20-40 minutes. Given that there were $3^7 = 2187$ possible parameter permutations, the estimated running time for 100 trials of each parameter permutation would be $2187 * 100 * 40 =$ 8,748,000 minutes, or approximately 6075 days. The allocated time to perform simulation trials was 1-2 months. Running 100 processes independently would put the number of days to just over 60, which was considered within acceptable bounds.

The decision to develop a web application to support and coordinate parallel simulations was based on preliminary experiments with the toy model to validate the decision tree and particle filtering methods. In this preliminary experimentation a MySQL database was used to collect and analyze results.

The experiment application was hosted on Amazon Web Services using a

2 GB Linux-based instance with a Linux, Apache, MySQL and PHP (LAMP) stack installed[4]. A front-end web page was developed using JavaScript, jQuery and PHP. Moderate attempts were made to secure the web server, which included restricted ports and enforcing Transport Layer Security (TLS) encryption via a certificate obtained from DigiCert, a multi-national certificate authority.

An executable Java file (.jar file) was available via a download link on the web page. Clients could download the executable and run multiple instances. Client hardware affected the quantity of instances possible per device as well as the running time for each simulation trial. Software version control was coordinated manually initially, with an automatic patch notification being added in a later release.

To prevent client data tampering, temporary simulation results were encrypted on client machines using AES-256 encryption until they were uploaded to the server.

Potential clients were volunteers solicited by social media and email. A preliminary email with the chair of the RMCC Research Ethics Board (REB) indicated that REB approval would not be required for this form of involvement by other people [18]. No compensation was provided to volunteers, as was indicated on a disclaimer on the main application web page.

Clients had the choice to run simulations anonymously or with an authenticated account obtained via email registration. The account option was provided to allow access to a “leaderboard”, which would log the statistics of computing power provided. There was no attempt made to capture internet protocol (IP) addresses or other identifying information beyond an account name[5]. The majority of trials (79%) were completed by anonymous clients.

Once a client had downloaded up-to-date software, the process to submit results was transparent to the client. When a simulation trial completed, the results were encrypted and queued in an upload folder. Periodically the software would scan for results in the upload folder, decrypt the file, convert to JavaScript Object Notation (JSON)[6] and upload to the server.

Initially parameters were obtained randomly from the 2187 possible parameter permutations, but was coordinated within the software as of version 3.00 to ensure an even number of trials would be generated for the selected permutations.

[4] https://bugbattleai.net

[5] This could be easily captured with modifications to server-level code.

[6] JSON is intended to structure information to facilitate encoding and decoding. For more information, see http://www.json.org/.

4.4 Summary

A preliminary experiment led to the conclusion to experiment with parameter permutation weights for the reward and utility functions instead of varying the population size and search depths. Even with this reduction in scope, the time to execute sufficient trails for an adequate analysis would be very long, so a distributed computing environment was created to facilitate multiple concurrent trials. The goal of the experiment was a uniform sampling of all parameter permutations.

5 Results

5.1 Introduction

This chapter is divided into the following major sections:

- Modified experiment design; and
- Experiment results.

5.2 Modified Experiment Design

After the initial week of experimentation, it was obvious that the rate of trials would not support the original goal of over two million trials. This was due to three main factors: server limitations, lack of available hardware and some trials taking considerably longer than preliminary tests. Initial problems with the database optimization and error logging led to a rate of approximately 6000 trials over seven days. These problems were corrected and the simulation was allowed to continue in the current state to obtain a better estimate on rate of completion.

When there were approximately 20,000 trials completed, the preliminary results were reviewed. There were 0 and 17 results for each parameter permutation. This was due to parameter permutation being randomly chosen within the application instead of being assigned by the server[1].

The original parameter space sampling was changed to select a distribution of 20 parameter permutations, but to assess for 1000 trials each. These permutations were selected from the sets with the most trials across a range of performance. At the time of choosing, the performance ranged from 100%

[1] This was intended to be a minor design feature of the web application to allow laptops to download the code and run without an internet connection, but the synchronization between the local cache and server was not implemented.

win rates down to 0%[2]. An approximately equal number of permutations were chosen from each grouping of 20%.

Preliminary results were reviewed again when the reduced permutation set had logged approximately 400 trials each. The best win rate was just over 50%, so a decision was made to increase the quantity of permutations to approximately 50. These permutations were chosen from a mix of the best preliminary performance and parameter weight patterns observed within the first 20 in-depth permutations[3]. The final number of chosen parameter permutations was 56, as six additional permutations were chosen after observing patterns with the energy and defense factors for both the reward and utility functions of the selected permutations.

5.3 Experiment Results

The main experiment collected just over 42,000 trials in the distributed environment. The total processing time was 21,041 h, or approximately 876 days of continuous processing if a single process had attempted to collect all the data. There were three main factors that affected the processing time for individual trials: client hardware, quantity of Dynamic bugs, and the ability of the Dynamic bug to find isolated and hidden Baseline bugs.

5.3.1 Data Collection

The logical organization of the database that captured trial data is given in Figure 5.1. Modifications made to this design to solve website performance issues include INSERT and UPDATE triggers on several tables[4]. These triggers were used for parameter permutation tracking and to augment entries in the "tblSimulationExecution" table, which contained the aggregate information for a simulation trial. The final database was approximately 1.1 GB, distributed as an SQL file on DVD as Annex D for this thesis. The SQL queries used to generate the data extraction in the figures throughout this section are also available on the DVD for Annex D.

[2]The win rates refer to the probability of the Dynamic bug defeating the Baseline bug.

[3]Generally, permutations with a high reward for energy were performing well. The focus was to select a mix of positive weights for energy and defense factors, both for reward and utility functions. Additionally, permutations that had non-zero weights for a wider range of factors were purposefully selected to ensure some exploration.

[4]A database trigger executes a segment of code when a pre-condition is met. For example, an INSERT trigger on table "X" would fire every time a record was added to table "X".

When there were approximately 6000 trials, client processes stopped logging trial data directly onto the database. This was done to keep the MySQL database more responsive. Aggregate data for the "tblSimulationExecution" table continued to be logged to allow summary statistics to be queried during the experiment execution. The full trial data was uploaded as a JSON text file and stored on the server with the filename based on the simulation name. When the data collection was completed, these files were parsed to add the full data to the database.

A coding error was discovered when preparing to extract the data from the JSON files. Each record in the "tblSimulationExecution" table has 1-to-many entries in the "tblSimulationTurn" table, which contains aggregate details on a turn-by-turn basis. The "tblSimulationTurn" table has 1-to-many entries in the "tblTurnStrategies" table, which contains a summary of strategies employed by all Dynamic bug instances on the given turn[5]. The problem was that only a single turn was parsed and recorded to the database for the first set of trials. The error was corrected before processing the approximately 36,000 trials worth of files on the server. An additional mitigating factor is that most of the lost data affected the initial exploration of the parameter permutation space, which means that at most 17 of 500 (3.4%) trials would be lacking the detailed strategy information for a single parameter permutation.

[5]This allows multiple layers of information at various performance costs. For example, all aggregate trial data was captured within 28,000 records, while the detailed turn statistics required approximately 950,000 records. Finally, exporting the turn information resulted in a comma-separated-value file that was 1.2 GB - larger than the entire database itself, because of how the query joined information. Each detailed strategy record was a summary of the observations on a given turn, because a brief analysis of the space required to capture the rewards for each strategy on each decision-action cycle would have resulted in a database in the 300-400 GB range. This would have been significantly reduced due to the reduction in parameter trials, so capturing detailed decision-action cycle information is a viable future experiment.

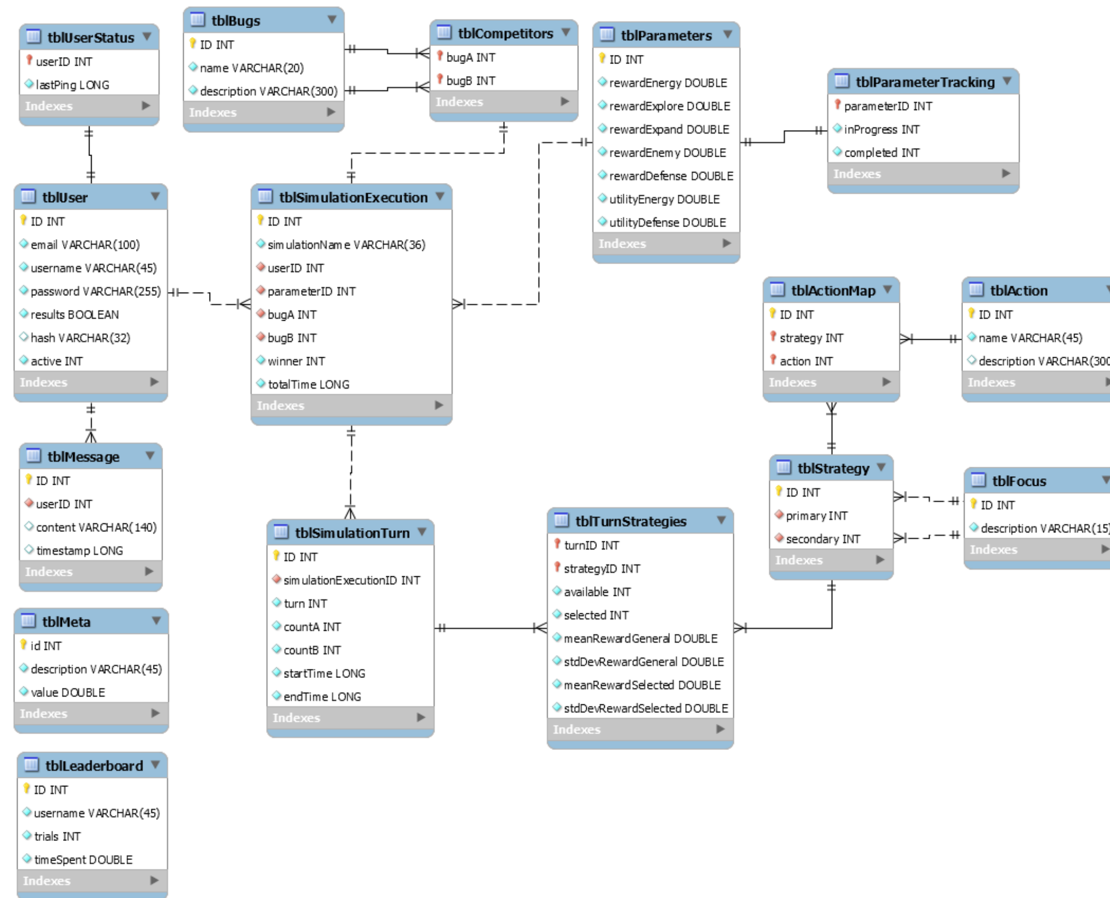


Figure 5.1: Data Collection Database - Logical Design

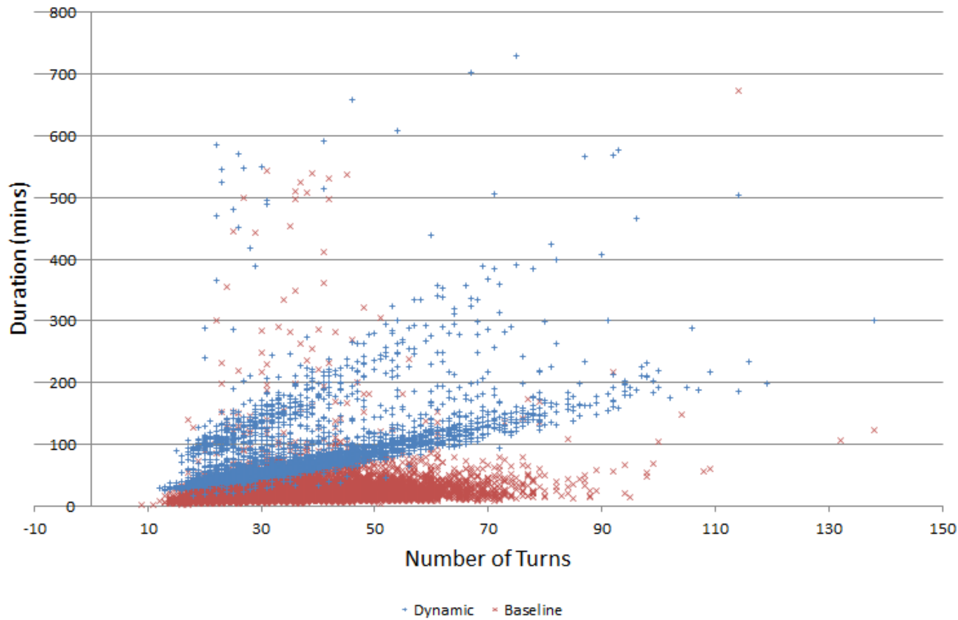


Figure 5.2: Duration of Simulation Trials

5.3.2 Processing Time

The overall simulation durations are provided in Figure 5.2. This graph captures all data from the selected trials with the exception of four outliers. One outlier was when the Baseline bug took 251 turns over 308 minutes to achieve victory. The other three outliers were cases where the Dynamic bug won and took longer than 800 minutes. The turn durations in these cases were 32, 51 and 144. The shortest number of turns was 9 and the longest was 251.

The average running times of the different parameter permutations are presented in Figure 5.3, in terms of quantity of simulation turns required. The aggregate data for the parameter permutations is included in Table 5.1. The graph is sorted by increasing win percentage, which is then superimposed on the graph. The error bars are the standard deviation of the calculated averages. There is no relationship between the win percentage and the turn duration required.

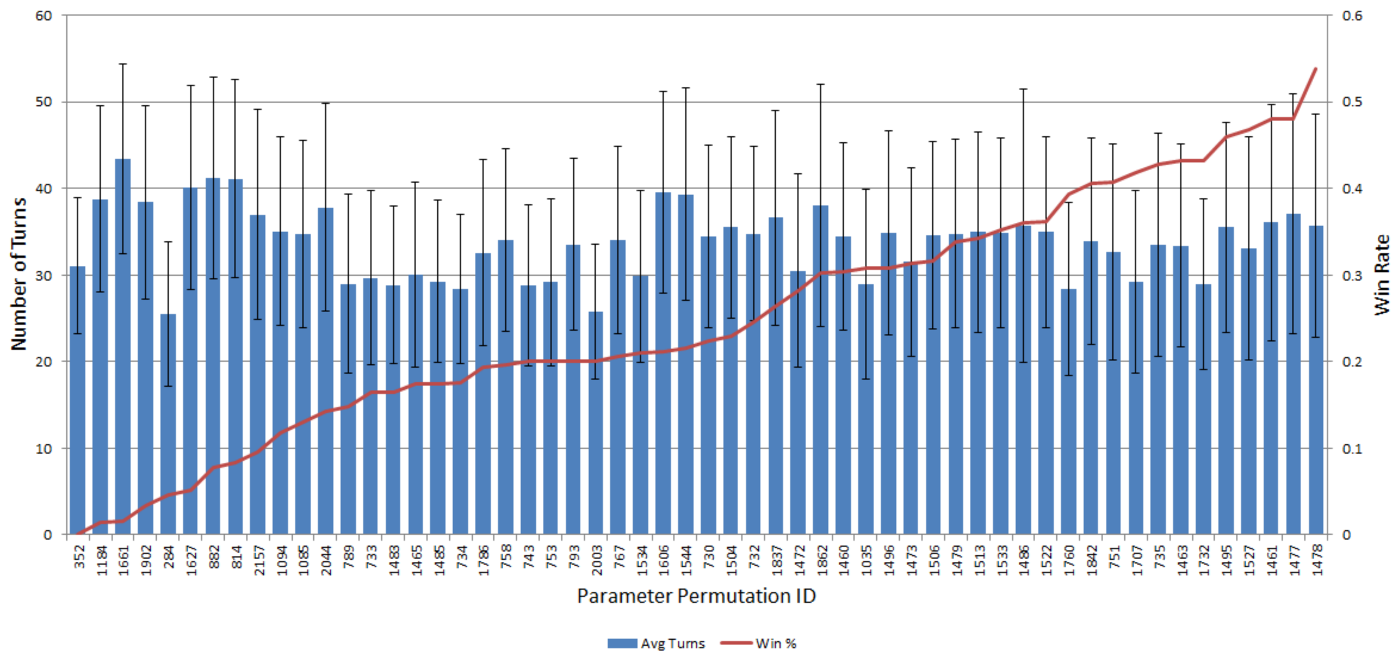


Figure 5.3: Average Turn Duration

When the effect of win percentage on processing time is considered, a clear pattern does emerge, as presented in Figure 5.4. This is largely due to the quantities of Dynamic bug in the simulation, which require considerably more processing time than the Baseline bug counterparts[6]. The error bars in Figure 5.4 are the standard deviation of the averaged trial processing time. The standard deviation is greater than the average in some cases, as a result of occasionally long trials[7]. From the trials for the selected parameter permutations, the shortest trial was 0.012 hours while the longest trial was 18.087 hours.

[6]Baseline bug processing time was not recorded in the experiment. Anecdotal observations estimate the processing time to be approximately 10 ms/ By comparison, the average (filtered) processing time for a Dynamic bug is 310 ms, see Figure 5.5 for more details.

[7]As an example, one of the parameter permutations with a low win rate of 8.4% but high standard deviation of 1,468,211 ms had an average processing time of 1,295,474 ms. This is the 9th from the left in Figure 5.4. The longest trial for this permutation was 16,881,788 ms, which was the result of the simulation taking 73 turns and the Dynamic bug winning. This permutation had only 33 trials that took over 3,000,000 ms. 70.8% of trials for this permutation had processing times between 500,000 ms and 1,300,000 ms

Table 5.1: Permutation Win Rates

Permutation ID	Win Rate	Reward					Utility	
		Energy	Explore	Expand	Enemy	Defense	Energy	Defense
1478	53.8%	2	0	0	0	2	0	1
1461	48.0%	2	0	0	0	0	0	2
1477	48.0%	2	0	0	0	2	0	0
1527	46.8%	2	0	0	2	1	1	2
1495	46.0%	2	0	0	1	1	0	0
1463	43.2%	2	0	0	0	0	1	1
1732	43.2%	2	1	0	1	0	1	0
735	42.8%	1	0	0	0	0	1	2
1707	41.8%	2	1	0	0	0	1	2
751	40.8%	1	0	0	0	2	1	0
1842	40.6%	2	1	1	2	0	1	2
1760	39.4%	2	1	0	2	0	1	1
1522	36.2%	2	0	0	2	1	0	0
1486	36.0%	2	0	0	1	0	0	0
1533	35.2%	2	0	0	2	2	0	2
1513	34.2%	2	0	0	2	0	0	0
1479	33.8%	2	0	0	0	2	0	2
1506	31.6%	2	0	0	1	2	0	2
1473	31.4%	2	0	0	0	1	1	2
1035	30.8%	1	1	0	2	0	2	2
1496	30.8%	2	0	0	1	1	0	1
1460	30.4%	2	0	0	0	0	0	1
1862	30.2%	2	1	1	2	2	2	1
1472	28.2%	2	0	0	0	1	1	1
1837	26.4%	2	1	1	2	0	0	0
732	24.6%	1	0	0	0	0	0	2
1504	23.0%	2	0	0	1	2	0	0
730	22.4%	1	0	0	0	0	0	0
1544	21.6%	2	0	1	0	0	1	1
1606	21.2%	2	0	1	2	1	1	0
1534	21.0%	2	0	0	2	2	1	0
767	20.5%	1	0	0	1	1	0	1
743	20.0%	1	0	0	0	1	1	1
753	20.0%	1	0	0	0	2	1	2
793	20.0%	1	0	0	2	1	0	0
2003	20.0%	2	2	0	2	0	1	1
758	19.6%	1	0	0	1	0	0	1
1786	19.4%	2	1	1	0	0	1	0
734	17.6%	1	0	0	0	0	1	1
1465	17.4%	2	0	0	0	0	2	0
1485	17.4%	2	0	0	0	2	2	2
733	16.4%	1	0	0	0	0	1	0
1483	16.4%	2	0	0	0	2	2	0
789	14.8%	1	0	0	2	0	1	2
2044	14.2%	2	2	1	0	2	0	0
1085	13.0%	1	1	1	1	0	1	1
1094	11.8%	1	1	1	1	1	1	1
2157	9.6%	2	2	2	1	2	1	2
814	8.4%	1	0	1	0	0	1	0
882	7.8%	1	0	1	2	1	2	2
1627	5.2%	2	0	2	0	0	2	0
284	4.6%	0	1	0	1	1	1	1
1902	3.4%	2	1	2	1	1	0	2
1661	1.6%	2	0	2	1	1	1	1
1184	1.4%	1	1	2	1	2	1	1
352	0.0%	0	1	1	1	0	0	0

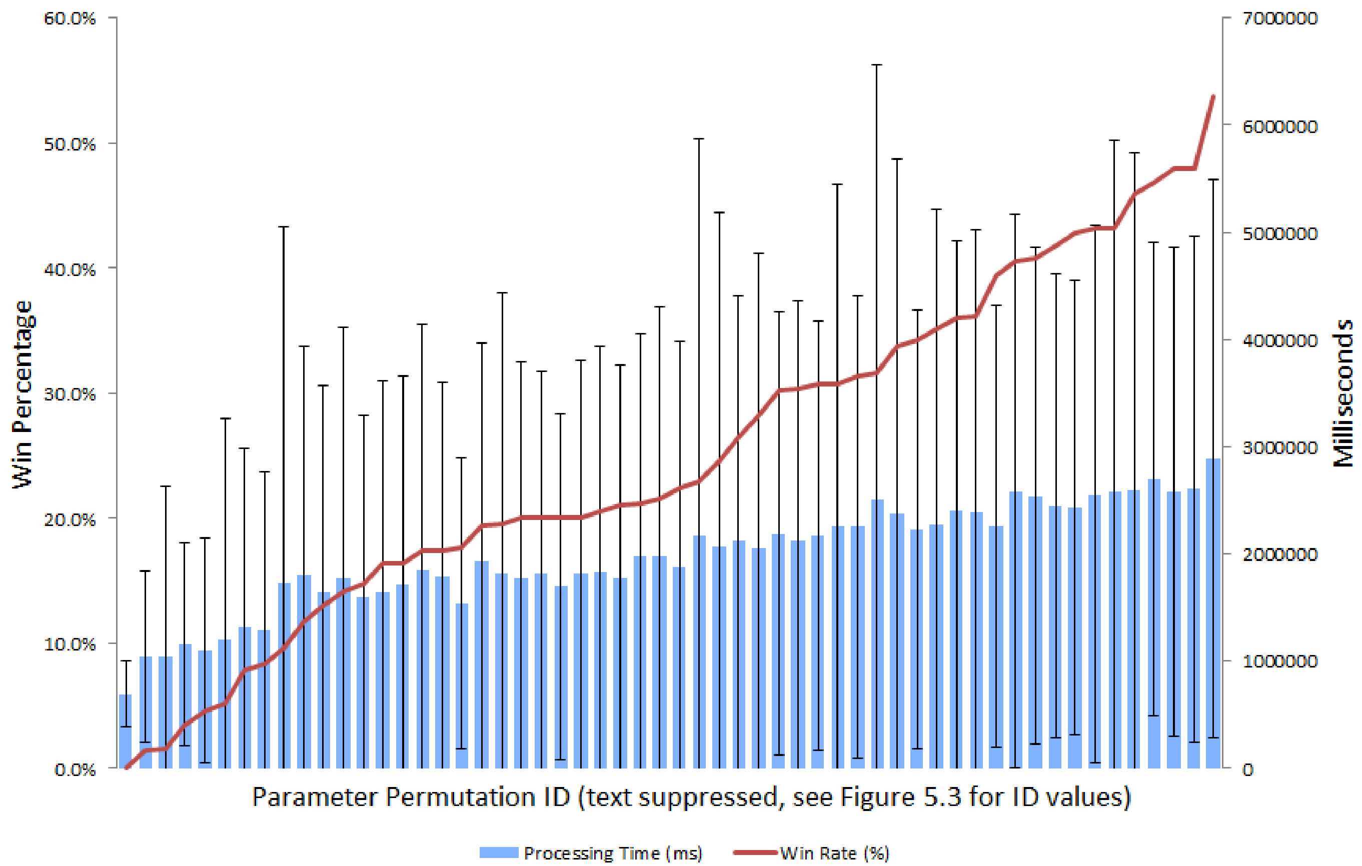


Figure 5.4: Average Trial Processing Time

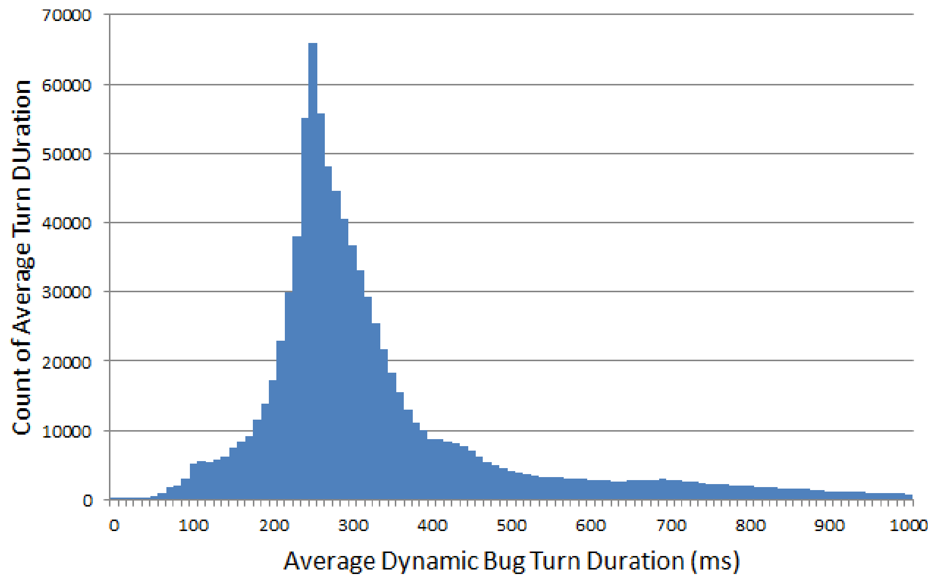


Figure 5.5: Dynamic Bug Processing Time

The estimated time for an individual Dynamic bug to execute a turn has a mean of 310 ms with a standard deviation of 173 ms. This was calculated by assuming an equal number of Dynamic bugs on each turn against the turn duration. 45,193 of 947,175 trials had a high processing time of over 1000 ms, which accounts for approximately 4.8% of the gathered data. This data was excluded from the calculations above[8]. Figure 5.5 presents the frequency of average durations less than 1000 ms.

An implied assumption in calculating the bug duration frequencies is that each bug takes an equal amount of time, which is known to lack accuracy due to physical design differences between the Scout and Worker variants. It also assumes that there are no other processing costs such as the Baseline bug or in-turn simulation control. The quantity of decision-action cycles a bug may execute on a turn is largely influenced by the quantity of cilia organs. Each actuation of cilia exposes a new environment and often provides additional

[8]Including all data results in a mean of 413 ms and a standard deviation of 7473 ms, which is due to the magnitude of the outlier data with values as high as 4,778,701 ms (1.3 h). The cause of the exceptionally high outliers is unknown. The most likely explanation is that these trials occurred before modifications to the client program to set fixed population size and search depth for the DDN algorithm. Since these were originally obtained from the web database and there were known performance issues with the database, this would be a viable explanation.

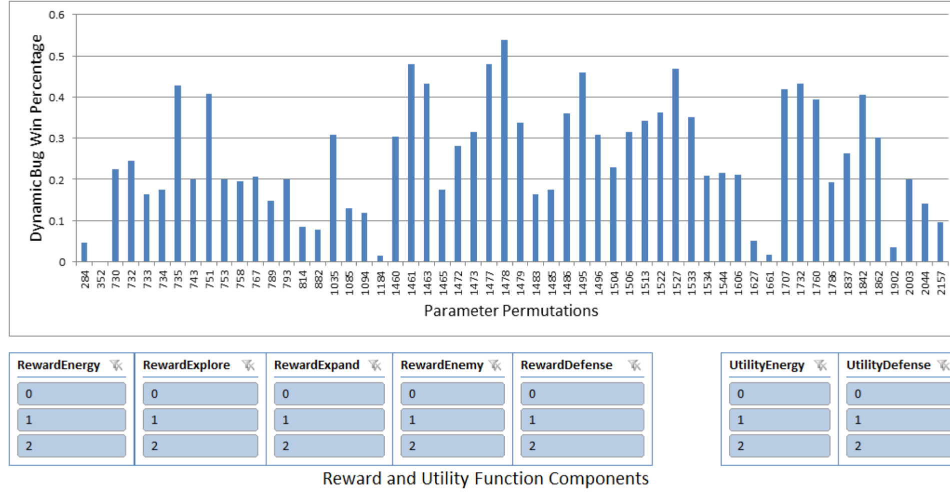


Figure 5.6: Dynamic Bug Win Rates, by Parameter Permutation

energy, which increases the ability of the bug to perform future actions. A bug may execute multiple decision-action cycles while stationary, however the energy of the bug will always decrease which limits the scope of potential actions. Since each decision-action cycle creates and solves a decision tree, the quantity of cycles performed by a bug on its turn has a significant effect on execution time.

5.3.3 Win Percentages

Win percentages of the selected parameter permutations are presented in Figure 5.6, which is a pivot chart with data slicers on the bottom[9]. The best performance of 53.8% and worst performance of 0% were achieved by parameter permutations given in Table 5.2. Table 5.2 also includes a row showing which filters to apply in order to remove the lowest performing permutations. No patterns emerged from further refinement of the filters. The results with the filters applied are represented in Figure 5.7.

[9]Data slicers are used to selectively filter data. When a slicer is filtering data, as in Figure 5.7, there is a filter icon highlighted red at the top-right of the slicer. Data points that do not match the criteria of this slicer are then filtered out. This can effect the available data for the other slicers, which may no longer have their full data set available. For instance, in Figure 5.7, 4 of 7 filters are applied but the remaining 3 filters still have their full data set.

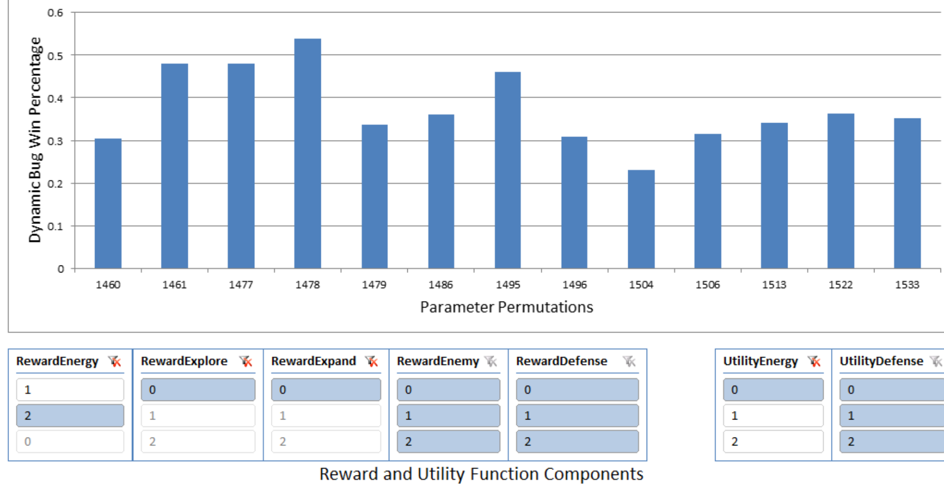


Figure 5.7: Filtered Dynamic Bug Win Ratios, by Parameter Permutation

Table 5.2: Dynamic Bug Performance - Reward and Utility Function Weights

Performance	Reward					Utility	
	Energy	Explore	Expand	Enemy	Defense	Energy	Defense
Best	2	0	0	0	2	0	1
Worst	0	1	1	1	0	0	0
Filtered	2	0	0	0-2	0-2	0	0-2

5.3.4 Strategy Selection

The most important factor that affects how a Dynamic bug performs is how it selects its strategies, which in turn corresponds to which low-level action is selected. While the action selection mechanism is based on solving the DDN, solving the DDN requires choosing one of the strategies with the best score. The mapping of strategies to actions was presented in Table 3.7.

Figure 5.8 shows the general bug populations by turn for each bug type. In these figures, the solid lines are the average bug populations and the error bars represent the standard deviation of the population on a given turn. The key trend to note is that the Dynamic bug tended to expand faster than the Baseline bug, but it tended to die off quickly once the initial plant resources were exhausted.

Isolating Strategies

Noting that there is significant variance in the populations, it becomes worthwhile to examine individual trials to discern patterns. The goal of this review

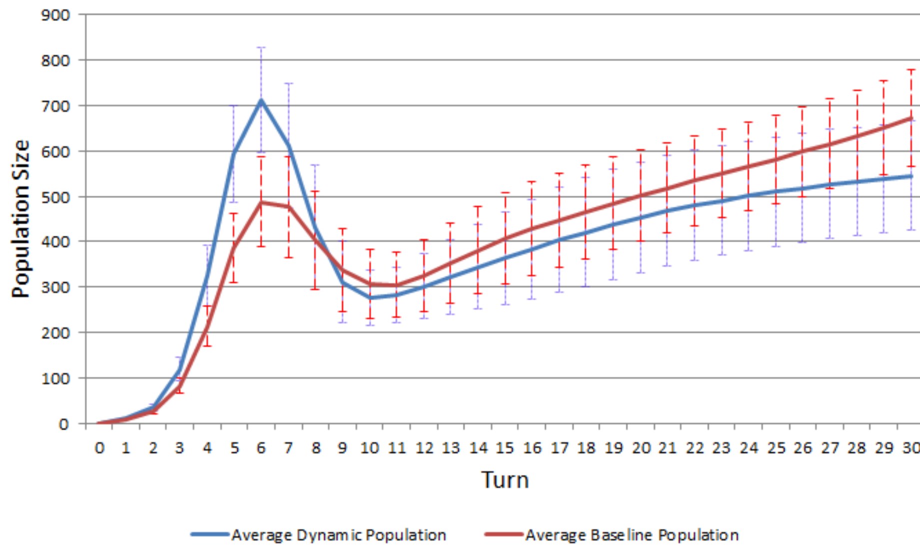
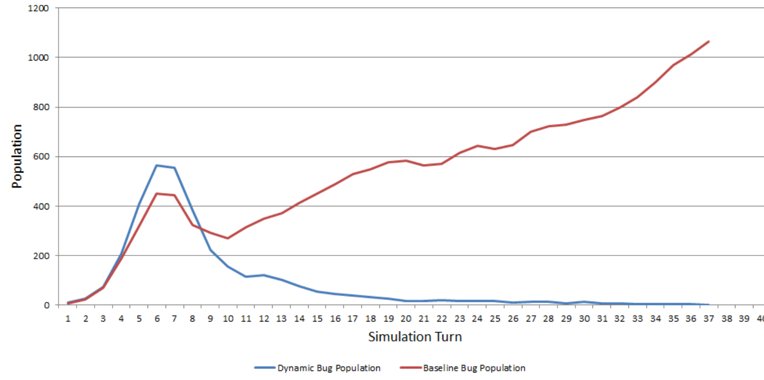


Figure 5.8: Average Bug Populations

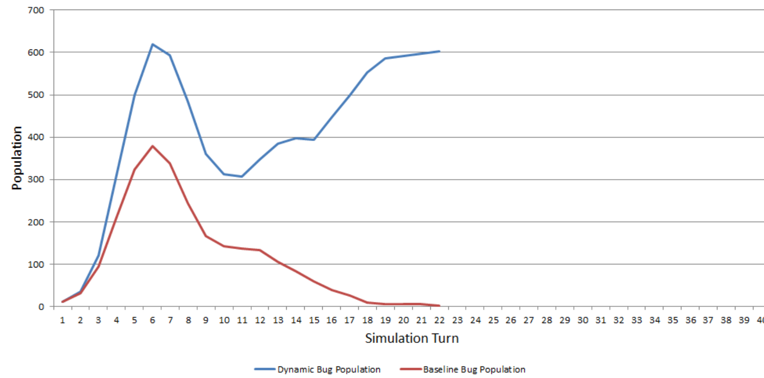
is to determine how to isolate the analysis of the strategy selection. A single parameter permutation was examined for general trends to guide the data extraction[10]. Figure 5.9a is one of a few common patterns when the Baseline bug wins. Figure 5.9b is a typical trial when the Dynamic bug wins. Lastly, Figure 5.9c exhibits a special case where the Dynamic bug is able to recover after falling behind the Baseline population. This last case is rare; of the 500 trials examined, this pattern was only easily discernible in 10 trials. The patterns that emerge from these general classes show that turns 3-10 are likely to yield the most descriptive behaviour.

A histogram of when trials are "determined", which was calculated using the first turn that the eventual winning bug obtained a population lead, is presented in Figure 5.11. The data is not shown for turn 1 because it accounts for 41.3% of all trials and significantly skews the histogram. The populations on turn 1 are highly suspectible to the initial plant populations (see Section B.1) which means that if the bug that eventually won had a few more plants randomly placed beside it, then it would likely have a population lead on turn 1. Based on the rapid growth in all scenarios, it is unlikely that this initial measure is a good indicator of performance. A more descriptive distribution emerges by changing the calculation of when trials were "determined" to the

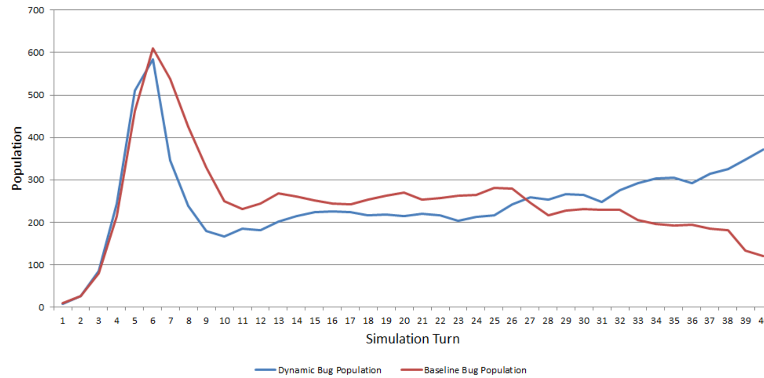
[10]The permutation with the highest win rate was chosen for this review.



(a) Baseline Win



(b) Dynamic Win



(c) Dynamic Recovery

Figure 5.9: Common Patterns for Bug Populations

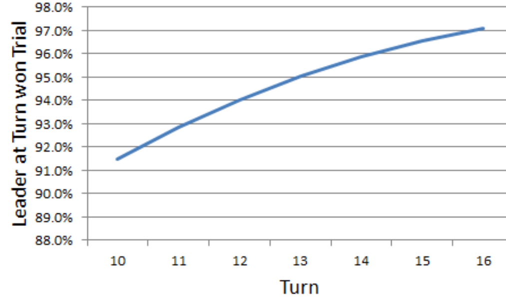


Figure 5.10: Percentage of Trials Determined by the Leader

state of the populations at turn t relative to the eventual winner. Approximately 91% of trials were won by the bug with the higher population at turn 10[11], which corresponds to the scenarios present in Figure 5.9a and Figure 5.9b. This increases considerably by turn 16, and is shown in Figure 5.10. This could be advantageous to consider for future work. Recall from Figure 5.2 that the vast majority of trials ran considerably longer than 16 turns.

At turn 10, there are two general cases for the unaccounted trials - either the pattern in figures 5.9a or 5.9b had not achieved dominance yet (but did soon afterward), or the trial exhibited the pattern in Figure 5.9c[12]. At turn 10, the case where there the Dynamic bug was losing but recovered to win the trial accounts for 3.1% of trials. The ability to maintain and re-establish the Dynamic bug population has an impact on success, as exhibited by Figure 5.12. Notably, the top five performing permutations accounted for approximately 35% of all recovery trials in this figure.

Strategy Details

Examining how the bugs prioritized action selection is crucial to gain insight into their performance. The population results above support the decision to limit strategy review to turns 1-10. Before reviewing strategy results, note that the DDN uses action filtering[13] when expanding the tree, so some strategies will be invalid for a single decision-action cycle. There is no action that

[11] 96.3% of trials were decided by turn 10 using the first definition of “determined”.

[12] There was potentially some overlap here - a “recovery” trial may have had a lead established by the turn plotted, in which case it would form part of the data in the graph.

[13] Actions are filtered on a sample-by-sample basis in the DDN. Since a single sample has defined values from the state space, based on probabilities drawn from the belief space, the filter can compare low-level conditions to see if an action is legal. For example, if the sample contains no enemy then all actions involving enemy targets would be filtered out.

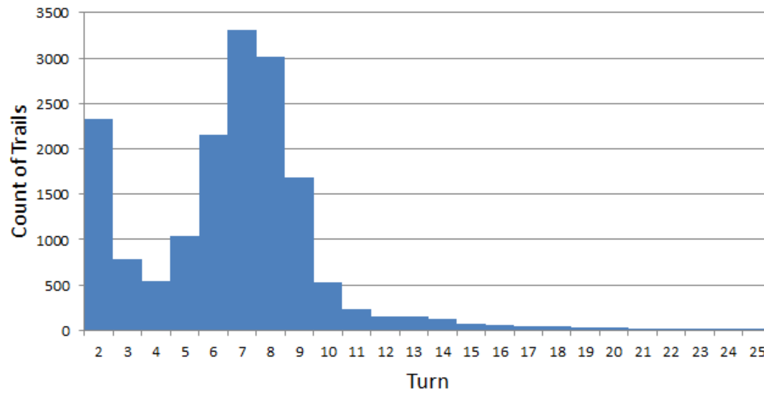


Figure 5.11: Trials When the Population Leader Won

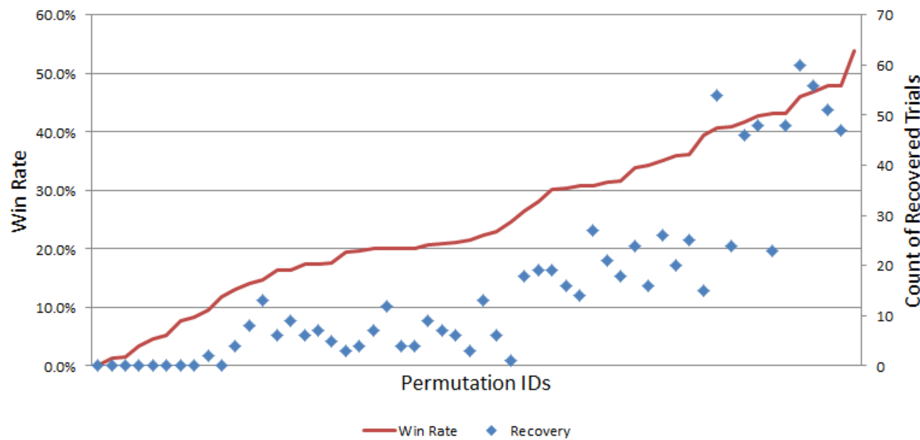


Figure 5.12: Recovered Trials at Turn 10

is always available, therefore it is impossible to determine the quantity of decision-action cycles performed on a single simulation turn. Further, the different physical characteristics of the Scout and Worker variants will produce considerably different action filtering[14]. The quantity of Scouts and Workers was not tracked but it would be a good piece of information for future experiments.

The remaining results in this section are obtained by extracting the average number of times each action is selected for each parameter permutation for

[14]Two obvious examples are movement and poisoning. Scouts are highly mobile but lack poison while Workers are immobile but make significant use of poison.

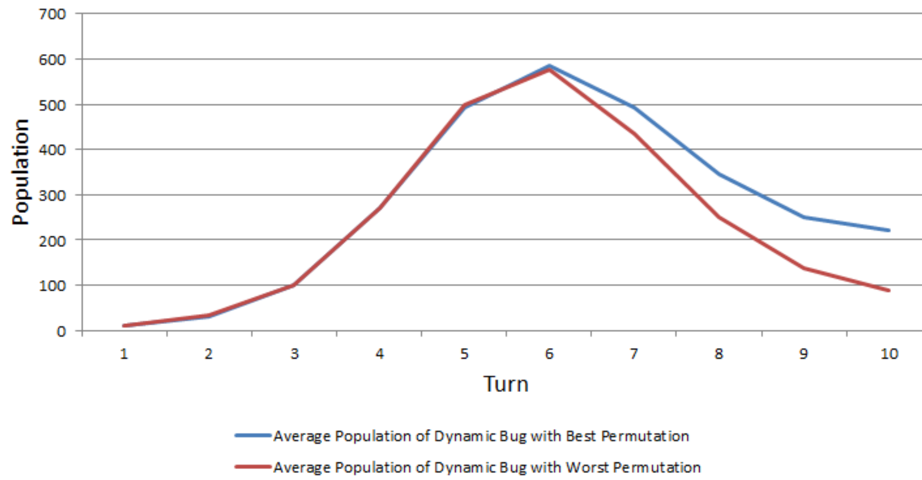


Figure 5.13: Differences in Average Dynamic Bug Population

turns 2-10[15]. Each record also contains the average population size for the Dynamic bug to allow a ratio based on the population sizes. For example, the population sizes between the parameter permutations with the worst and best win rates is presented in Figure 5.13. While the initial turns are almost identical, in the later turns the better-performing permutation has an average population that is approximately twice as large.

Figure 5.14 presents the distribution of actions selected on a typical turn for a Dynamic bug. This particular distribution came from turn 5, for the permutation with the best performance. The general trend of choosing movement toward the best energy or expansion instead of the other choices is consistent across all permutations for most turns. This is likely due to the simulation world being largely unexplored during these initial turns. Filtering out these most-common actions for this sample yields the action distribution presented in Figure 5.15. The distribution of these remaining actions does change turn-by-turn and between different permutations. The evolution of action selection by general category is presented in Figure 5.16, which highlights the general progression from the exploration phase to consolidation. General categories

[15] Note that the beginning turn here is turn 2 vice turn 1 as discussed in the preceding paragraph. The reason for this is a coding error in the data logger where results for turn 1 were not logged properly. The exact cause of the error was not determined with certainty, but it was likely a problem when processing the JSON script in PHP due to occasional errors that were noticed and suppressed for "turn 0". This was readily apparent when reviewing detailed data as the action SPIT-POISON-TO-KILL had high occurrences on turn 1 when that situation should have been almost impossible.

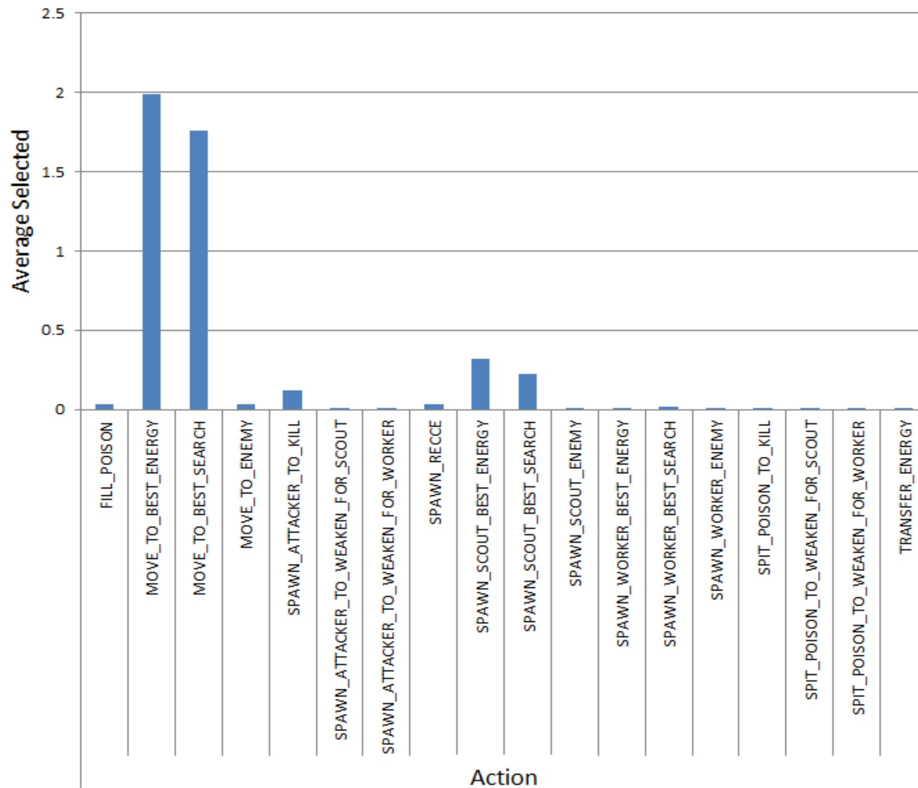


Figure 5.14: Typical Action Distribution on a Single Turn

are defined by the intended purpose of the action. Figure 5.17 views the same data isolated by parameter permutations, restricted to the worst and best performing permutations. While the trends for both permutations are consistent with the general case, there are some differences in prioritization. In the earlier turns the lower-performing permutation attempts to kill enemies more often which is expensive in terms of energy. In the later turns the lower-performing permutation attempts to weaken enemies more often. Intuitively, weakening an enemy for a subsequent spawning action would better harvest the adversary energy. Given the low performance of the permutation, it is likely that the Baseline bug has established defensive counter-measures by this stage which would cause the newly-spawned bug to die, resulting in an overall weaker Dynamic colony.

Figure 5.18 displays the distribution for the best and worst performing permutations for the most common actions. Some trends can be obtained when

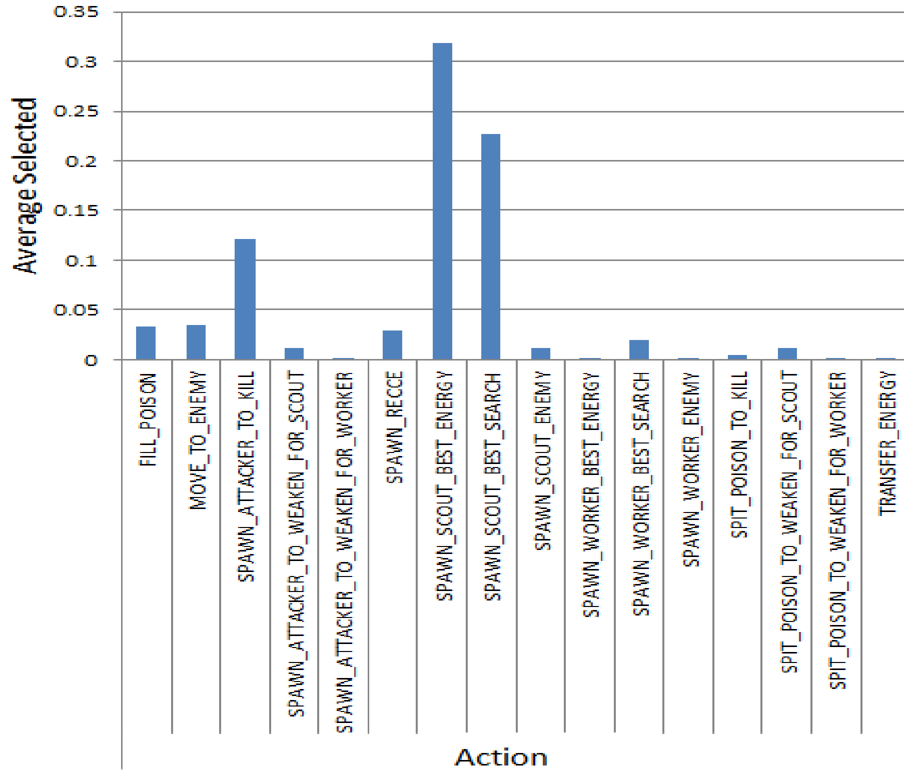


Figure 5.15: Typical Action Distribution on a Single Turn - Filtered

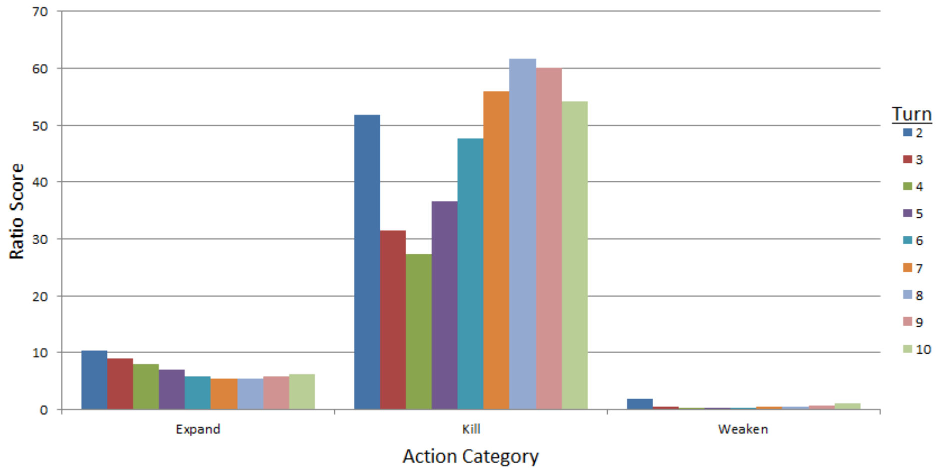


Figure 5.16: Generalized Action Evolution

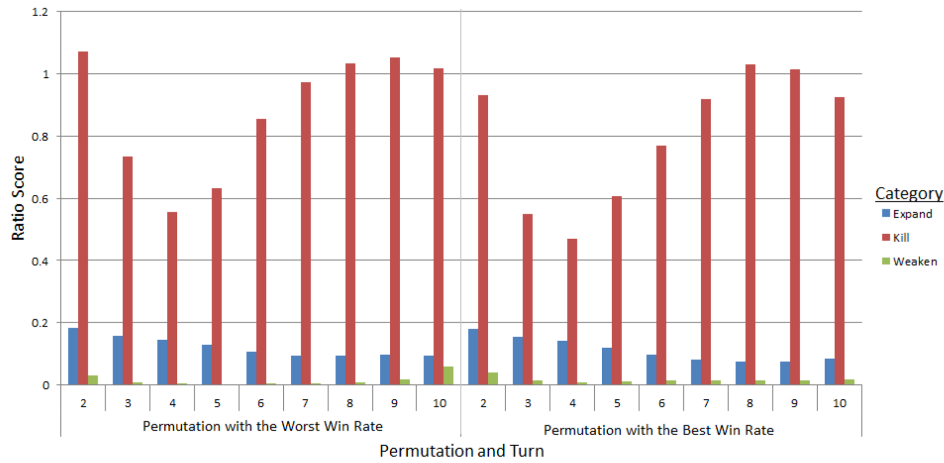


Figure 5.17: Comparative Action Evolution

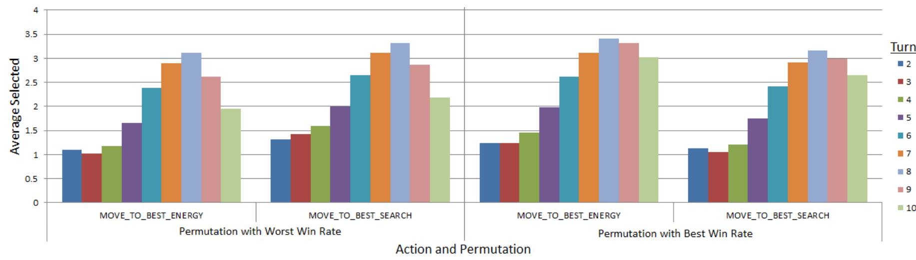
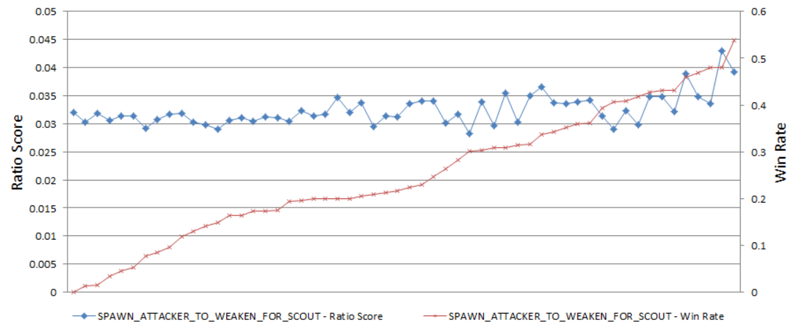


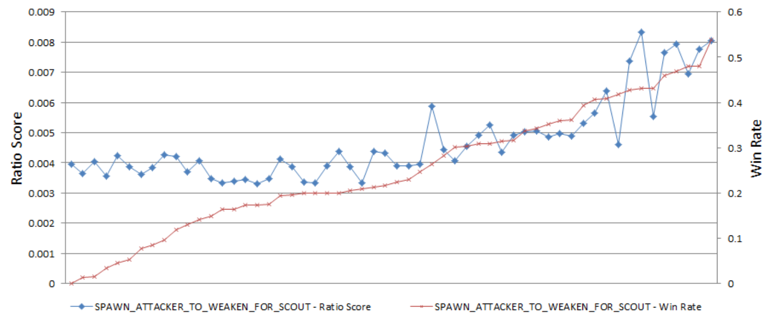
Figure 5.18: Comparison between Permutations for Selected Actions

viewing the distribution information from the individual action perspective. In this particular Figure, the priorization between energy and exploration is apparently reversed for the two permutations. The worst-performing permutation prioritizes exploration over exploiting energy. This suggests a potential reason why the best-performing permutation had a higher population in later turns, as higher per-bug energy levels would permit a more stable colony. Bug energy levels were not recorded during simulation trials. Adding bug energy for each decision-action cycle to future experiments may permit deeper review into the decision-making process and the effect on aggregate performance.

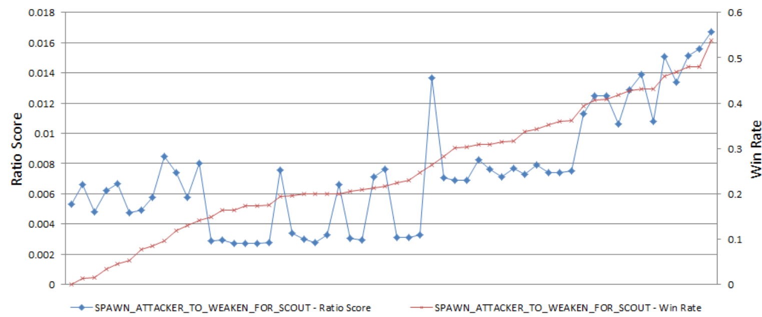
Expanding and sorting the permutations considered allows further trends to be examined. Figure 5.19 presents the data for turns 2, 4 and 7 for all permutations for the action where the Dynamic bug uses the attack bug to weaken an enemy to allow a subsequent spawning action. Permutations are



(a) Attack Action - Turn 2



(b) Attack Action - Turn 4



(c) Attack Action - Turn 7

Figure 5.19: Progression of an Attack Action

sorted by increasing win rate with ratio scores superimposed. As the turns progress, a pattern emerges where the high-performing permutations use the action more frequently.

Table 5.3 presents the major relationships between turns and actions cho-

sen. Actions not present in the table have scattered results. The graphs are rotated at a variety of angles to highlight the relationships in the respective graphs. The "Reward Ratio" in these graphs was the ratio of the quantity of the selected action against the Dynamic bug population at the given turn to provide an approximation of the frequency of choosing the action. Very large values such as MOVE-TO-BEST-ENERGY imply that the action would be chosen several times during a turn whereas low values such as SPAWN-SCOUT-ENEMY imply the action is rarely chosen. This is not a perfect measure as the ratio of Scout to Worker variants will have a large impact on the quantity of decision-action cycles which affects the quantity of decisions that can be made in a given turn.

Unsurprising, MOVE-TO-BEST-SEARCH, MOVE-TO-BEST-ENERGY, SPAWN-SCOUT-BEST-ENERGY and SPAWN-SCOUT-BEST-SEARCH are very frequent at the beginning of a simulation as the environment is largely unexplored. The probability of choosing these actions decreases significantly as the simulation progresses. The pattern is largely consistent across all parameter permutations, with the exception of the spawning actions. The lower-performing permutations are only about half as likely to spawn Scouts at this stage.

SPAWN-SCOUT-ENEMY, SPIT-POISON-TO-WEAKEN-FOR-SCOUT and SPAWN-ATTACKER-TO-WEAKEN-FOR-SCOUT are nearly identical relationships. The delta between higher-performing and lower-performing permutations peaks around turn 7 and quickly stabilizes by turn 10. At turn 7 the higher-performing permutations are approximately three times as likely to use these actions. However, the reward ratios for these actions are exceptionally small so they are rarely used.

MOVE-TO-ENEMY and SPAWN-ATTACKER-TO-KILL are far more likely as the simulation develops. While SPAWN-ATTACKER-TO-KILL is nearly consistent across permutations by turn, MOVE-TO-ENEMY is only about half as likely in higher-performing permutations.

5.3.5 Simulation Observations

The probabilities generated from bug observations were not recorded due to the sheer volume of data. The simulation code, available on DVD as Annex D, provides a graphic user interface to explore generated probabilities on a trial-by-trial basis. Two key assumptions made when generating probabilities were use of fixed values or normal distributions. Fixed values were used for centrally-tracked allies and plants because the information was either known with certainty or it was extremely likely to occur. Normal distributions were

used when generating aggregate probability information for a single location. There is some risk in applying a normal distribution here, based on the low number of observations. An exponential or Bernoulli distribution may be better to use here, but due to time constraints on implementation the normal distribution was chosen.

There are several other visual debuggers created in the simulation code designed to verify the advanced techniques discussed in Annex B. These debuggers need activation in the source code to function.

5.4 Summary

The scope of the original experimentation plan had to be adjusted due to technical difficulties and limited time. This led to a reduced quantity of parameter permutations.

The extremely high computational cost was underestimated, but a review of the processing times shows that there is a range of performance. As the Dynamic bug performed better the trials took longer. Win rates were significantly affected by the reward sub-function weights. Adjusting these weights was shown to have an effect on the low-level actions chosen.

Table 5.3: Action Trends

Action	Trend	Remarks
MOVE-TO-BEST-ENERGY		Very similar to MOVE-TO-BEST-SEARCH.
SPAWN-SCOUT-BEST-ENERGY		Very similar to SPAWN-SCOUT-BEST-SEARCH.
SPAWN-SCOUT-ENEMY		Virtually identical to SPIT-POISON-TO-WEAKEN-FOR-SCOUT and SPAWN-ATTACKER-TO-WEAKEN-FOR-SCOUT.
MOVE-TO-ENEMY		SPAWN-ATTACKER-TO-KILL is similar, except higher win rates are more likely to spawn an attacker than lower win rates.

6 Analysis

6.1 Introduction

The analysis of the experiment is spread across several facets:

- The performance of the Dynamic bug relative to the Baseline bug within the simulation;
- The computational cost of executing the DDN algorithm; and
- The challenges of assessing agent performance.

6.2 Dynamic Bug Performance

The hypothesis suggested that an effective strategy would emerge by treating the Bug Battle simulation as a DDN. Although the overall win rate of the Dynamic bug was approximately 20% for all trials, the five permutations that had the best performance had an average win rate of 48.5%, while the best permutation had a win rate of 53.8%. While this was not quite the majority of trials, it was moderately effective. As discussed in Section 3.4.1, the implementation of the reward sub-functions had minor flaws that may have degraded performance. This will be discussed further in Section 6.2.1.

One of the tenets of agent-based models is that agents should be autonomous to handle the complexity that exceeds the bounds of direct intervention. The Baseline bug was designed using approximate heuristics based on informal experiments within the simulation framework. These rudimentary experiments were by no mean systematic; they were small tests to probe the effects of the various framework mechanisms. The initial tests were typically performed in isolation. They hinted at the underlying probabilities, but trials were limited to basic A-B testing[1]. Understanding the framework complexity facilitated the development of the Dynamic bug.

[1] A-B testing is comparing two versions of a product or system to determine which one has better performance.

The ability of the Dynamic bug to manage this complexity was mixed. It had considerable success in the unopposed environment. The ability to expand in the initial environment is critical to establish a population baseline from which to grow while denying the same opportunity to the adversary. The Dynamic bug significantly out-performed the Baseline bug in initial growth, evident by Figure 5.8. The problem that the Dynamic bug encountered was when the initial resources were depleted. All variants would die-off to a degree, which is expected because there is not enough energy to sustain the population[2]. However, it was clear that some permutations managed the combat environment better than others. Figure 5.9c was an excellent example of this as it showed a rare ability to recover. The challenge with designing this performance will be discussed further in Section 6.4.

A deeper analysis of the dominant strategy may have led to a better experiment design. The dominant factor is rapid and efficient use of energy, yet many parameter permutations had a weight of zero for this factor. This failure was spectacular - the total win rate for all trials that met this condition was 0.8%. The importance of energy is the only factor that is immediately obvious in this regard, and energy is a contributing factor in 4 of 7 sub-functions, so a deeper analysis was not considered necessary. The revised selection of parameter permutations in Section 5.2 effectively provided this coarse refinement based on initial trial results.

6.2.1 Reward Function Implementation

As was suggested by the preliminary experiment in Section 4.2.1, the selection of the reward and utility functions has a significant effect on the simulation outcomes. This was readily apparent by the win rates demonstrated in Figure 5.6. This matches the established literature on reward functions (or fitness functions for genetic algorithms) as discussed in Section 3.4.1.

The implemented reward functions differed from the intended relationships. From Equation (3.1), each reward sub-function had an intended effect based on aggregate patterns observed[3]. The implementation of these functions had only a cursory verification based on code tracing.

[2]This happens with the Baseline bug as well, although it tends to be a bit slower.

[3]These patterns differed by sub-function. The energy sub-function had increasing value up to an energy of 2000 based on an x^2 function at which point it should have transitioned to a linear function. By contrast the exploration sub-function had globally diminishing returns based on the assumption that a heavily explored environment would not require significant dispersion.

The result of the rushed verification was that the energy and defense sub-functions had incorrect coding. The error in the energy sub-function is assumed to be small as the relative difference is minor, values in the range of (1.0,2] are mapped to 2. The range due to the implementation of the defense sub-function is approximately [0,7.5] vice the intended [0,2][4]. This impacts the expected outcome of the experiment but it does not compromise the overall approach. The extended range of the defense sub-function suggests that defensive measures has a more important role than initially assessed, given that the top-performing trials all have large defensive weights. Many of the permutations with win rates in the range of 10% to 20% also have large defensive weights, so the presence of high defensive weights is not exclusive to success.

The high computational cost of executing simulation trials is an obstacle to evaluating variations of reward functions. A more effective approach may be to develop a testing data set of belief space distributions to enable rapid (isolated) testing of reward functions to validate their implementation. The simulation experiments would still be required as the coverage of the belief space for function validation would be infinitesimally small by necessity. "Isolation" means only a single decision would be considered, which is a consequence of the sheer scope of the belief space. Evaluating the outcome from many single states without a simulation framework may be viable, but once multiple steps are considered there would be too many scenarios to manage.

Another benefit of building a testing data set is the ability to incrementally add features. When the reward sub-function for the enemy factor was designed, the intent was to reward the bug for killing or weakening the enemy. The implementation rewarded the action selection only, not the actual effect[5]. There was enough information in the simulation to detect and reward the effect directly. Discovering these scenarios during development may lead to expansion of the encoded features. It may also justify additional data collection. For example, it would have been relatively easy to develop statistics for attack outcomes, would could have been used to reinforce or extinguish the behaviour.

[4]Both cases should be relatively rare because the implementation error only impacts energy levels beyond the creature energy cap.

[5]While the bug would always be successful in killing or weakening uncloaked enemies, the bug may die as a result of defensive damage.

6.3 Computational Cost

The simulation experiment was very expensive to run from a computational perspective. Based on Section 5.3.4, executing each trial until one bug becomes extinct is not necessary. The challenge is determining a suitable metric to signal the end of a trial. It is possible for bug populations to oscillate as was observed in the "recovery trials" (see Figure 5.9c), so fixed durations are not ideal. There is a point where the opponent cannot recover. Isolating this relationship would allow a significant reduction in the time required to run a similar experiment[6]. In general it would be a good idea to recognize system conditions to design suitable termination triggers.

Optimized code is another way to reduce the computational cost. Some techniques such as variable caching were used to avoid re-calculating fixed costs that were obvious in the design phase. Optimizing code requires a potentially high investment in programmer time, so the potential increase in execution efficiency should be weighed carefully. Some of this analysis can be done by using application profiling for features such as memory use or method invocations. Since this experiment was intended to be run as a distributed environment there was limited application of code optimization.

The most significant factor for the high computational cost is the algorithm selection. The branching factor of the DDN produces exceptionally large trees, especially for large search depths. A search depth of two was used in the main experiment because the preliminary experiment had similar or degraded performance for deeper searches at a much higher processing cost.

One way to reduce the large cost due to branching is to apply the research on HDDNs discussed briefly in Section 2.3.5. The intention-based actions provided to the DDN planner were partially based on the HTN concept of HLAs. Table 5.3 shows that some actions appear to be sequenced after each other, such as SPAWN-ATTACKER-TO-WEAKEN-FOR-SCOUT followed by SPAWN-SCOUT-ENEMY. The sequence of actions was not part of the recorded data so this pattern cannot be confirmed with certainty. Encoding these sequences in a HDDN as an HLA would reduce the span of the DDN which has been shown to have a significant impact on computation cost [29].

[6]Using the overly-simplistic cut-off of ending at turn 16, accounting for approximately 97% of trials, the overall experiment time for the selected parameters would have been reduced from 15,360 h to 4412 h, approximately 29%. The cut-off assumes that all trials exceeding 16 turns are capped at 17.0 minutes, which is the average duration of the 256 trials that took 16 turns or less.

6.4 Assessing Agent Performance

Agent-based models are by nature designed using simple rules with the expectation that patterns of behaviour will emerge. While designing the low-level rules is straight-forward, it is difficult to predict the effect on the emergent behaviour. Heuristics developed from experience and preliminary testing guided the design of the Dynamic bug actions and their efficient tactical implementations. The representation of the belief networks was based on the same heuristic analysis. The observation-tracking system employed by the HQ (see Section B.3.7) was built to support the belief space representation that was designed.

While the design process was incremental, the representation of statistics was fixed which led to some desirable information being lost. For example, the parameter permutation with the worst performance attempted to kill enemies significantly more than the parameter permutation with the best performance at turn 10. Based on their relative performance, the likely effect was that the spawned bug died due to defensive counter-measures. As was warned in Section 2.2.3, it is important to note that the bug was behaving rationally. There was limited encoding in the belief state about the presence of poison for an enemy. It was considered initially but was not implemented due to the difficulty in validating without a robust test set. The reliance on heuristics reinforces the earlier suggestion to develop a test set of scenarios.

There were other statistics that may have been beneficial to track that only became apparent when performing the analysis. The difference in quantities of Scout and Worker variants is one example, as it would have allowed deeper analysis into the behaviour of the planner for different physical models. Similarly, tracking the bug lifespan (ideally by variant) would influence the selection of the utility function. The model assumed an infinite planning horizon, which is very unlikely. Application debugging and observing trials suggests that bugs die frequently even in an unopposed environment. This implies that a finite horizon may be more appropriate, which would require modifications to the utility function. As alluded to in Section 6.3, tracking the sequence of decision-action cycle outcomes would likely yield common sequences as candidates for HLAs, which would further improve computational efficiency.

There were some statistics that were recorded but not used in the analysis. The average and standard deviation of the scores for the reward function were recorded for each action, but the high variance caused by the aggregation negated their usefulness as there were too many confounding factors to establish any trends. The decision to record aggregates instead of individual

scores was based on space estimates for the final database. A better approach would have been to record individual scores for a smaller experiment, as a deeper understanding of the effect of the reward function on action selection may have been possible. Using a small experiment would keep the results to a manageable size and could be iteratively applied for refinements to the reward function. This was not done due to time constraints.

6.5 Summary

While the performance of the Dynamic bug was not as strong as anticipated, several potential improvements were observed for both the execution time and the design of the reward and utility functions. A test set of belief states could be used to validate reward functions before extensive simulations and the experiment would benefit from being conducted as a series of incremental sub-experiments. This would guide selection of statistics in the belief space representation to gain insight into the emergent properties of the agent-based model.

7 Conclusion

The application of DDNs to Bug Battle yielded improved performance relative to the Baseline bug. This result can be replicated for the single reward function that achieved a success rate greater than 50%, but the experiment suggests that further parameter tuning would yield improved results.

This performance came at a high cost; simulation trials using the DDN approach are too slow to be feasible for the Bug Battle competition. The parameterization of the best Dynamic bug implementation would need to be examined further to create an improved bug that could execute within the time limits of the Bug Battle competition.

The majority of reward functions examined were unsuccessfully in meeting the aim of this work. This reinforces established literature which indicates that selection of a reward or utility function has a significant outcome on agent decision-making [20]. The trade-off of flexibility of decisions against high computational cost was apparent by the results in Section 5.3.2.

7.1 Contributions

The main benefit of this work was providing an example of how to apply and solve DDNs for a particular problem. There are two large challenges in this type of application: selecting an appropriate DDN model and choosing appropriate reward and utility functions. The experiments run for this thesis demonstrate a method to test and refine the selection of the reward and utility functions.

The implementation of the distributed environment (see Section 4.3) is a flexible architecture that can be reused for other experiments that have a high computational cost. The code is available for re-use in Annex D. The main elements required to re-use are the website code and the *bugbattle.distributed* package in the source code. The *bugbattle.distributed* package is essentially an application wrapper that communicates with a website. Both portions require

some modification for other experiments.

The Java code provides several other features for research into Bug Battle or DDN in general. These include:

- Flexible DDN implementation
- Event-driven architecture for simulation events
- Graphic User Interface (GUI) debugging controllers for spatial environments

7.2 Future Work

It is likely that extending the DDN approach used in this research to a HDDN may speed up the simulation execution. This has two benefits; incremental refinements to the reward and utility functions would be easier to test and the execution speed may be viable for competition. The core idea that facilitates a HDDN speed-up is reducing the breadth of the decision trees used to solve the DDN.

An improvement that would facilitate additional trials is the inclusion of a premature termination trigger. Although the simulations are "won" when an opposing bug is eliminated, in most cases the eventual winner has a population lead early. The vast majority of trials had the winner in the lead by turn 16, yet most trials took considerably longer for the last remaining bugs to be eliminated. Developing a metric other than elimination of opposition would avoid these scenarios. The metric should not be based purely on population leader at turn X. Based on the population patterns in Figure 5.9, a suitable metric may be a combination of a minimum turn and a population delta by a relative or absolute amount. This was not tested in this research due to time constraints.

Also considered for future work is increased variable-tracking to guide insight into the emergent properties of the system. Detailed tracking of bug decisions, quantities of Scout or Worker variants and bug lifespans would facilitate development of the HLAs discussed above.

A Glossary

Acronyms

ABM Agent-Based Model. 4–6, 19
ACO Ant-Colony Optimization. xi, 84, 85

BN Bayesian Network. 16–19
BRIC Block-like Representation of Interactive Components. x, 9, 10

DBN Dynamic Bayesian Network. 18, 19
DDN Dynamic Decision Network. ix, 2, 16, 18–21, 26–28, 30, 33–35, 49, 51, 54, 64, 67, 70, 71
DN Decision Network. 18

GUI Graphic User Interface. 71

HDDN Hierarchical Dynamic Decision Network. 16, 19, 30, 67, 71
HLA High-Level Action. 16, 67, 68, 71
HTN Hierarchical Task Network. 15, 16, 19, 67

JSON JavaScript Object Notation. 38, 42, 56

MDP Markov Decision Process. 11–14, 17

POMCP Partially Observable Monte-Carlo Planning. 15
POMDP Partially Observable Markov Decision Process. 2, 13, 15, 16, 19, 20, 34, 95

r.v. random variable. 16–18
REB Research Ethics Board. 38
RL Reinforcement Learning. 10, 11, 14

SQL Structured Query Language. 3, 41

TD learning Temporal-difference learning. 14, 15

B Bug Battle Simulation

The purpose of this annex is to give sufficiently detailed explanation of the Bug Battle simulation to understand what the agents are attempting to achieve in the experiment. The annex is divided into several sections for ease of reading:

- The main simulation framework, which explains how the major elements of the simulation logic functions.
- Bug organs, which provides a listing and analysis of the different organs available in the simulation.
- Advanced techniques, which describes several methods to implement tactically advantageous behaviours for a bug.

B.1 Simulation Framework

The Bug Battle simulation is a Java project. The 2015 version of Bug Battle was used for this research[1]. This discussion assumes the use of "agent", "bug" and "creature" interchangeably.

The simulation user interface consists of two parts: bug selection (Figure B.1) and simulation execution (Figure B.2, note that green dots are plants). To prepare a new simulation instance the process is as follows:

- the environment is cleared of all creatures;
- plants are randomly generated with an initial uniform probability of $\rho = 0.12$; and
- three instances of each participating bug are created and placed in the environment.

B.1.1 Simulation Logic

The simulation is multi-threaded but only in the sense that the graphic user interface and simulation logic run on separate threads. The simulation logic

[1] Minor glitches are fixed between versions, core functionality is not modified.

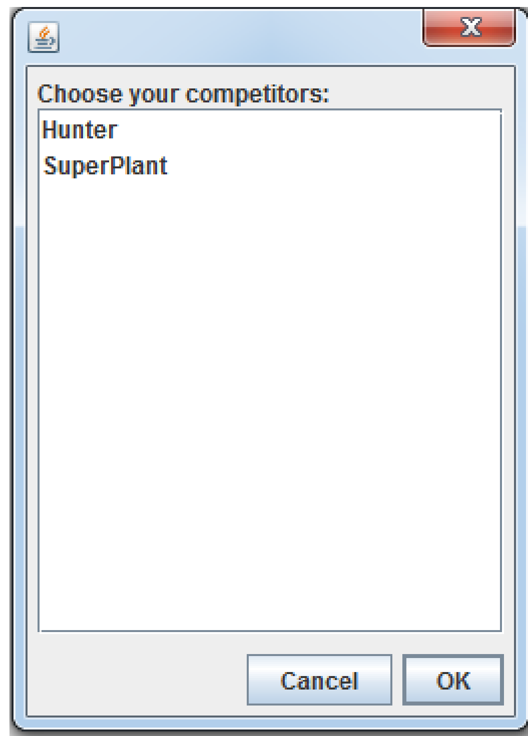


Figure B.1: Bug Selection

consists of a loop that executes turns. The simulation loop is presented in Figure B.3.

Simulation Loop

There are a few significant elements to note from this loop in terms of the simulation. First, only the creatures that exist at the start of the simulation turn can receive a turn which means that new bugs created during the turn cannot execute their turn logic until the next simulation turn [2]. Secondly, the list of creatures is traversed using fast enumeration. Order of execution is not guaranteed to be consistent, so turn-order bias should be minimal. Lastly, the probability of open ground spontaneously becoming a plant is adjusted based on the current number of plants in the environment. Even in the case of bugs that expand as quickly as possible to deplete the initial plant population

[2] It is possible for a newly-created bug to receive a turn immediately by explicitly invoking its doTurn() method, but this is not allowed in the simulation rules.

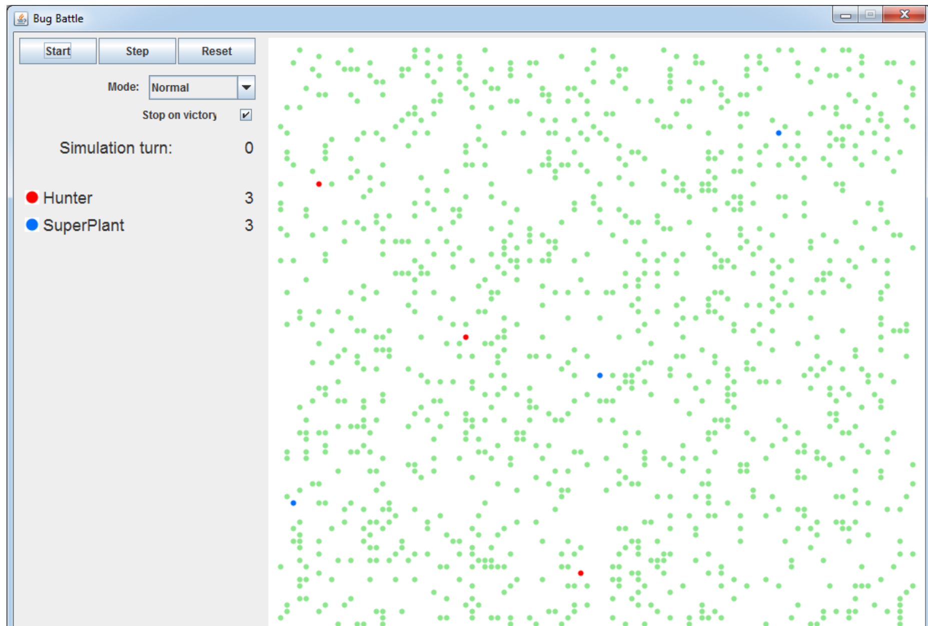


Figure B.2: Simulation Execution

there will almost always be plants in the simulation due to this conditional probability.

For the agents themselves, Figure B.3 presents three key methods[3]. The maintain() method reduces bug energy based on a baseline maintenance cost as well as the maintenance costs of all current organs. It also resets organs with limited actions per turn (cilia in particular) and deducts the cost of cloaking, if cloaked. The doTurn() method is where the agent can define its behaviour. Finally, the capStrength() method ensures that the agent strength does not surpass a maximum of 2000 energy.

Although not in the diagram, when agents are instantiated, they execute an initialize() method before their first doTurn(). Initialization typically adds initial organs, but bugs can minimize energy loss in the maintenance phase by delaying adding organs until their doTurn() is invoked[4].

[3]The logic in this figure is represented as a sequence diagram in the Unified Modelling Language.

[4]A trade-off exists here if attempting to spawn directly onto an enemy. Allowing a bug to delay organ creation leaves it with higher energy to win combat, so the energy cost to the parent is reduced. The downside of this delayed initialization is that an enemy may destroy the un-initialized bug before it gets a turn, which would give the attacker a better energy return. This is assumed to be rare as the energy cap is not applied until the end of the first

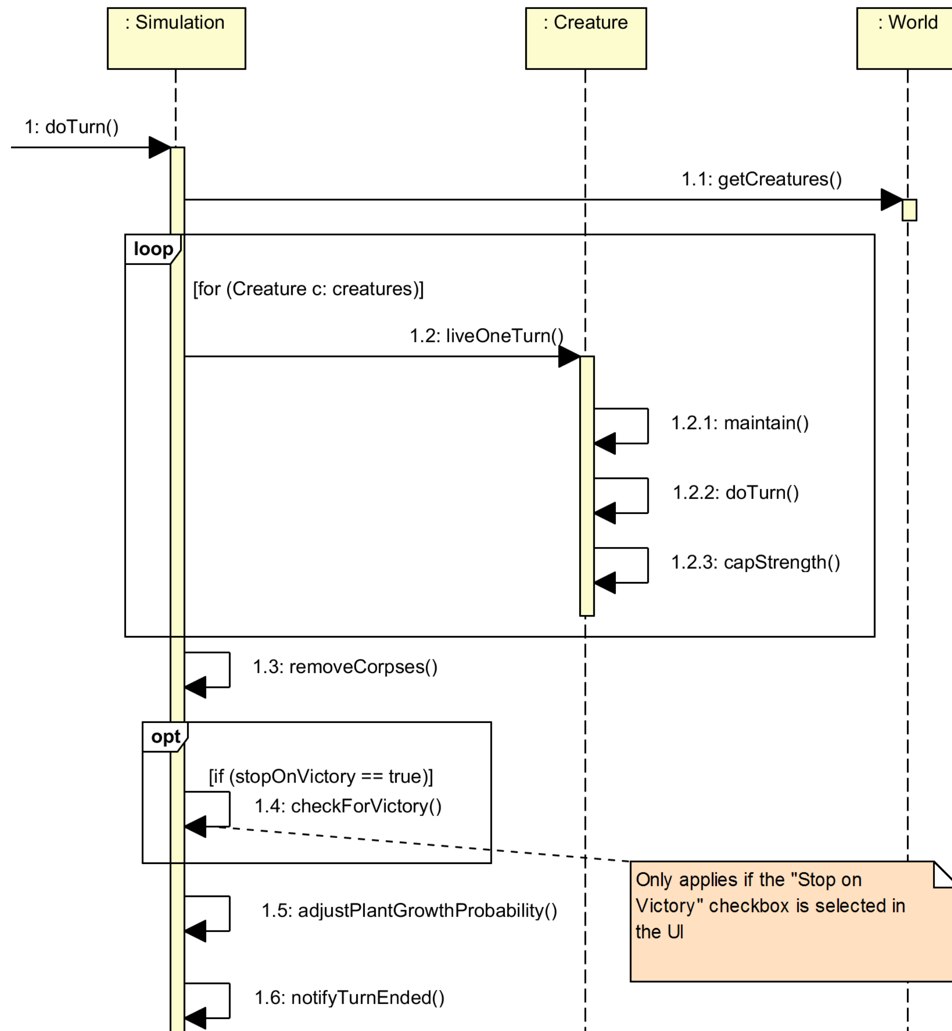


Figure B.3: Simulation doTurn() Sequence Diagram

Since the energy cap is not applied until the end of the turn, it is possible (and common) that an agent will have more than 2000 energy during the turns. This is crucial to allowing some of the techniques discussed in Section B.3.

Simulation Combat

Simulated combat occurs immediately whenever two agents occupy the same location in the environment. The actual energy level of each creature is compared. If the attacker's (the creature moving onto the cell) energy is greater than or equal to the defender's energy, the attacker wins, otherwise the defender wins. The winning agent absorbs all the energy of the losing agent and then defensive damage (if any) inflicted by the losing agent is applied to the winner. This can result in the winner being killed as a result of combat. Defensive organs are discussed in Section B.2, and techniques to optimize defensive damage to kill adversaries is discussed in Section B.3.10.

Note that there are two ways to initiate combat, either moving onto a cell using cilia or spawning onto a cell using a budder. There are two techniques discussed in Section B.3 that use the spawning approach.

B.2 Organs

Each bug can have zero-to-ten organs attached. Table B.1 provides a summary of the different costs for each organ[5]. The sections that follow provide a description of the capability in the physical model that the organ adds as well as suggestions for viable employment. While most organs have parameters that could be adjusted in the code, these settings are assumed to be fixed. Note that creatures have a base maintenance cost of 100.

doTurn() invocation for the bug. Un-initialized bugs will frequently have far more energy than the energy cap permits.

[5]The organs are introduced by their simulation representation, e.g. "OrganName". When they are referenced later in the text this name is decomposed to the more readable "organ name" representation.

Table B.1: Bug Battle Organs

Name	Creation Cost	Maintenance Cost	Use Cost	Purpose	Remarks
Budder	50	5	100	Create new bugs	Unlimited uses
Cilia	100	10	20	Move	Max 1 move/turn
Cloaking	500	10	100	Become invisible	No cost to toggle
CreatureTypeSensor	100	10	2	Detect type	Unlimited uses
EnergySensor	100	10	2	Detect energy	Unlimited uses
LifesignSensor	50	5	1	Detect presence	Unlimited uses
PhotoGland	500	-150	N/A	Generate energy	Passive
PoisonGland	500	20	Variable	Create/spit poison	Max reservoir of 1000
PoisonSensor	50	5	1	Detect poison	Unlimited uses
Spikes	200	5	N/A	Defensive damage	Passive

B.2.1 Budder

Budder is an abstract class that must be sub-classed. The purpose of this is to force creature-specific sub-classes to provide an implementation of the method that specifies what kind of creature(s) should be created when the sub-class is used. Using the budder creates a new instance of some type of bug in a given direction that is adjacent to the parent.

One approach that provides significant flexibility is to have the budder conditionally determine what type of creature to create. This allows for bug variants to be custom-tailored to specific situations. Specialized techniques that use this idea are discussed later in Section B.3.

B.2.2 Cilia

Cilia allows the bug to move up to a maximum of 1 cell per turn. However, there is no restriction (other than the organ limit) on the number of cilia that may be added to a given creature. Recognizing this flexibility is essential to developing a bug that can rapidly and efficiently explore the environment.

B.2.3 Cloaking

Cloaking renders the bug undetectable to all sensors. Since sensors are the only framework-provided mechanism to gather information about the environment, this can render the bug invisible. The negative consequence of this mechanism is that the bug is also invisible to friendly instances of bugs, so it is very possible that friendly instances will kill each other. A special technique to mitigate this effect is discussed later in Section B.3.9.

Despite appearing to be invisible, the bug is still subject to normal combat rules. Balancing use of limited energy with situational invisibility could be a useful component for an effective secondary bug variant. However, the high energy cost of the organ and its employment preclude cloaking from being an effective choice for a main bug variant[6].

[6] This implication mostly follows from not having sufficient energy to explore the environment rapidly to exploit plant energy. A preliminary primary design using cloaking was explored, but it was too slow to be competitive with the Baseline agent. The Baseline agent killed the cloaking agent by covering all the locations in the environment before the cloaking agent could establish self-sufficiency, regardless if it knew the cloaked agent was there or not.

B.2.4 CreatureTypeSensor

When used, this sensor reports the apparent type of creature that resides in the requested adjacent cell (given by the Java class name). Cloaking can fool this sensor.

Using this sensor aids determining friendly bugs, plants, open ground (dirt) and enemy bugs. The information gained for a single organ position and fairly low cost is quite beneficial. However, since plants have a well-defined energy pattern depending on the number of turns they have been alive, this sensor could be replaced with another organ if necessary as long as the agent has an energy sensor. Given that plants begin with fixed energy and assuming they live a maximum of 10 turns, the energy values for plants only occupy 0.55% of the values from [0,2000][7].

B.2.5 EnergySensor

When used, this sensor reports the apparent amount of energy present in the requested adjacent cell. Cloaking can fool this sensor.

This sensor is very useful to guide decisions on whether or not to engage in combat, and how to select targets. As noted above, the energy sensor alone is sufficient to determine a creature type between open, plant or enemy with high probability. Accounting for friendly bugs is discussed later in Section B.3.

B.2.6 LifesignSensor

When used, this sensor reports if the requested adjacent cell contains some form of creature. Cloaking can fool this sensor.

Although the costs to add and use this sensor are lower, adding this sensor occupies a valuable position in the physical model with limited information gains.

B.2.7 PhotoGland

This organ is used to generate energy (which is why it has a negative maintenance cost). It is the only way bugs can generate energy other than combat.

Since the only ways to gain energy are via combat or this organ, it is very common for bugs to have several of these organs. An alternate strategy is to aggressively seek combat by using several cilia.

[7]Limiting the expected plant life to 10 turns is a reasonable assumption in a rapid expansion environment. The Baseline agent easily exploits the initial plants within 10 turns.

B.2.8 PoisonGland

This organ adds an offensive and defensive capability to the bug. Once this organ is added, bugs can spend energy to fill a poison reservoir (to a max of 1000 units). The reservoir is expended in one of two ways: by spitting poison onto an adjacent cell or dealing defensive damage if the bug is defeated in combat. The damage multiplier is 4-to-1[8], which makes this organ the most effective way to reduce enemy energy levels.

The poison-spitting can be used in either attacking an enemy directly, or spitting poison onto an empty cell. One technique that can be used is to spit poison onto an enemy such that it is weak enough to defeat in combat, and then to move onto it using cilia. Spitting poison onto an empty cell (which creates a poison drop) will damage any creature that moves onto it, which can form a protective barrier. However, this technique is not particularly effective because the strength of the poisoned creature is not considered, so a creature with 1 energy could negate a poison drop with 1000 energy invested into it. Further, the energy in a poison drop dissipates each turn.

A comparison between defensive organs is discussed in Section B.3.10. This section also covers advanced techniques on how to effectively use poison glands.

B.2.9 PoisonSensor

When used, this sensor reports if the requested adjacent cell contains a creature that is apparently poisonous. This could be either a poison drop or a bug with a poison gland. Cloaking can fool this sensor.

Similar to the lifesign sensor, the poison sensor has limited use.

B.2.10 Spikes

Spikes are an organ that cause defensive damage to be applied when the bug with the spikes is defeated in combat.

A comparison between defensive organs is discussed in Section B.3. Spikes have limited use for primary designs as they are only useful when the bug dies.

B.3 Techniques

This section contains advanced techniques developed independently by the author. A primitive version of the energy transfer technique was used in the

[8]This means a full reservoir of 1000 units is capable of dealing 4000 units of damage.

graduate course for which credit was previously received. A more advanced version of this transfer mechanism is combined with adaptive spawning and the "HQ" (see Section B.3.7) to direct energy where it is perceived to be needed most. Probabilistic exploration was also recognized informally during the graduate course, although no analysis of the probabilities or related statistics was performed in the course.

These techniques are assumed to be basic knowledge for the purposes of the thesis.

B.3.1 Probabilistic Exploration

To prevent bugs from aimlessly wandering, perhaps in circles, it is beneficial to assign a default direction for a bug to travel in lieu of more information. The goal of this technique is to encourage exploration.

Plants are initially spawned in the environment with a uniform probability density of $\rho = 0.12$. Thus, the probability that a bug is initially placed beside a plant is given as:[9]

$$\begin{aligned} P(plants \geq 1) &= 1 - P(plants = 0) \\ &= 1 - (1-\rho)^8 \\ &= 0.64 \end{aligned}$$

Assuming that the environment is symmetric, the choice of how to assign a default direction can be reduced to choosing diagonal directions or vertical/horizontal directions. With horizontal or vertical movement three new cells are explored whereas diagonal movement explores five new cells. The respective probabilities of finding a plant with each option then becomes:[10]

$$\begin{aligned} P(plants \geq 1|\text{diagonal}) &= P(plants = 1) + P(plants = 2) + \ldots \\ &\quad + P(plants = 5) \\ &= 5 * (0.12)^1(0.88)^4 + 10 * (0.12)^2(0.88)^3 \\ &\quad + \ldots + 1 * (0.12)^5(0.88)^0 \\ &= 0.472 \end{aligned}$$

[9]Recall that each cell is considered to have 8 neighbours.

[10]Note that sampling each cell becomes a "trial" with a true or false outcome, so the calculations are based on the binomial distribution.

$$
\begin{aligned}
P(plants \geq 1|\text{vertical}) &= P(plants = 1) + P(plants = 2) + P(plants = 3) \\
&= 3 * (0.12)^1(0.88)^2 + 3 * (0.12)^2(0.88)^1 \\
&\quad + 1 * (0.12)^3(0.88)^0 \\
&= 0.319
\end{aligned}
$$

This means that diagonal movement should be preferred. The preferences for initial bugs are randomized amongst the four diagonals. When later bugs are spawned, their movement preference is orthogonal to their parent to encourage faster dispersion.

Bugs should be created with sufficient energy to explore their environment so that they have a reasonably high chance of finding food.

B.3.2 Frequency-Based Exploration

Alternative methods to the simplistic dispersion in Section B.3.1 were considered, using Ant-Colony Optimization (ACO) [6, 7] and influence maps [28, 35] as inspiration. The general idea of ACO is that agents explore the environment and deposit "pheromones" where they move, with the pheromones dissipating over time. When agents are choosing where to move, locations with higher pheromones are more likely to be chosen. In influence mapping, a grid (or mesh) is laid over the environment. Key features are assigned scores which are highest in the grid they occupy, and propagate to neighbouring cells subject to some fall-off rate. This allows algorithms to assess the value of a grid quickly.

Given the aim to encourage dispersion, the idea is to reverse the ACO concept. The idea is to maintain a grid of "search scores". Each cell in the environment is initialized with the value $\tau_{\text{UNEXPLORED}}$. When a friendly agent moves onto a cell, the value is set to 0. On each simulation turn, the values in the cells decay (increase) by τ_{DECAY}. To allow agents to consider shared regional information beyond their local sensor ranges, pre-defined search patterns of length r are considered. As movement in any direction is equally valid, the search patterns are constructed such that each direction considers an equal number of cells. An example with $r = 3$ is presented in Figure B.4. When choosing which direct to prioritize exploration, the sum of each grid is considered. The highest search values (least explored directions) are held in a data structure. Once all directions have been evaluated, the best direction to explore is taken from the data structure[11].

[11] Note that directions with friendly agents are removed from consideration to prevent unwanted combat. This may not be strictly necessary as the probability that those directions have the highest search score is very low.

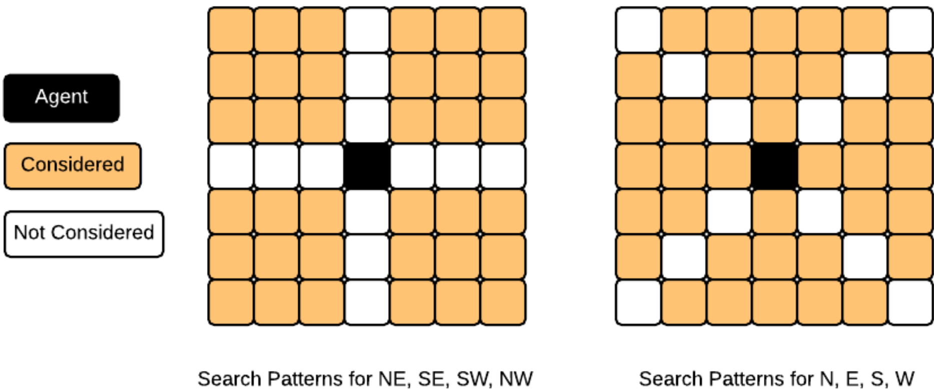


Figure B.4: ACO-Inspired Search Patterns

Preliminary experiments with $r = 4$ found dispersion behaviour of the baseline agent to be similar to the probabilistic exploration method. This is largely because the agent only relies on this approach to move when no other information is available. If plants or enemies are nearby then they are targeted first.

B.3.3 Adaptive Spawning

Adaptive spawning is advantageous based on a cursory analysis of the initial environment and the limitations of the physical model for the bugs. Initially there are many plants and very few enemies in the environment. As the simulation progresses, the bug density (potentially) quickly increases, to the point where the organs added for early exploration become ineffective for colony consolidation. Further, there are cases such as Section B.3.5, Section B.3.6 and Section B.3.9 where entirely different conceptual bug designs are warranted.

Aside from the special sections noted above, two ways to address adaptive variation are to change the organs added dependent on the limited system state information they have access to or to create entirely different bugs (which are still considered allies). This research uses the latter approach, as it is easy to manage the complexity by varying levels of abstraction[12].

[12]From an object-oriented perspective, common behaviours can be built into a base class and specialized variants can extend the base class to achieve their particular purpose.

B.3.4 Multiple Spawning

As alluded to in Section B.3.3, some bug variants should prioritize environmental exploration at all costs. In the expansion phase of the simulation, it may be possible that a bug has sufficient energy to generate many children. Spawning multiple children in this phase results in faster exploitation of plant resources which in turn results in a more rapid transition to consolidation while denying adversaries the same opportunities.

Multiple spawning is generally not considered effective in the consolidation phase of the simulation, because positions to spawn tend not to have plants for rapid expansion. Furthermore, the spawned bugs need to have higher energy levels to survive contact with adversaries.

B.3.5 Energy Transfer

When a bug is surrounded by allies and it has generated surplus energy beyond the 2000 energy cap, this energy would be wasted unless it can be transferred to an ally. This can be achieved by spawning a basic bug with up to $x - 1$ energy onto the bug targeted to receive the energy, where x is the energy of the target bug. If x is sufficiently small, this basic algorithm can be looped until the desired amount of energy is expended.

Choosing how much energy to transfer between bugs is debatable. Retaining only the amount required to survive the maintenance phase will push the maximum amount of energy where it is needed, but it also makes the colony interior susceptible to penetration. There is no apparent optimal strategy on which to base this decision.

If the relative position within a colony is known (see Section B.3.7), there are three main approaches to deciding where to transfer this energy. Firstly, the easiest mechanism is to always go in the same direction that will have the effect of one edge of the colony being very strong and the opposite edge being very weak. Secondly, the agent could detect which neighbour(s) will lead to the closest edge based on the assumption that this location will need energy the most. Lastly, the colony could tweak the second behaviour to include adaptation to threats. In this behaviour the colony would detect, classify and prioritize where energy should be sent based on where attackers have struck[13].

[13]This behaviour was not implemented. With rapid expansion it is rare for large colonies to form, so the benefit of this approach was considered to be negligible.

B.3.6 Kamikaze Strikes

Methods of offensive action in the simulation are limited to movement, spawning and spitting poison. Movement is effective only when the current agent has more energy than the target. Spawning a standard bug variant onto a target faces a similar limitation, but because of the use cost of the budder organ, the energy thresholds are even lower. Poison may be effective if coordinated, but the poison gland requires a position in the limited organ hierarchy.

A hybrid approach is the most energy-effective form of attack; it retains the flexibility of not adding a poison gland to the normal build order. This approach spawns a specialized kamikaze bug designed to lose combat but inflict significant defensive damage on the target. To maximize the defensive damage, the kamikaze bug should have no energy left after creating defensive organs. Targets can be weakened or killed with this method. In the case of weakening the target, the original bug can then move onto the target to gain its remaining energy.

Choosing how to build a kamikaze agent depends on the energies of the source bug and the target. Optimal energy use to inflict a given amount of damage is discussed further in Section B.3.10. Choosing how to select targets is left to the agent designs.

B.3.7 Headquarters (HQ)

In the original BugWars framework, bugs could access their absolute position in the environment. As long as each instance passed this information to a higher coordinating mechanism, colony-wide behaviour could then be coordinated. The conversion to Bug Battle removed this mechanism, so bugs no longer have any knowledge of their location in the environment, at least in the exposed portion of the framework available to competitors.

However, this locating mechanism can be created using relative position tracking and principles of reconnaissance (recce). Initially there are a set number of bug instances in the simulation. Each bug in the simulation is assigned a unique identification number, for tracking in various data structures. Each of the initial bugs assumes an initial local position of (0,0), and a tracker identification number to indicate their group ownership. When bugs are spawned, they are added to the same group as the bug that spawned them. By tracking births, movement and deaths, relative positioning can be developed independently for each group.

In order to synchronize the independent groups into a single collective, the bugs make extensive use of message-passing via the observer pattern and

top-level observer object (the "HQ") to coordinate events. Since bugs only have energy sensors, they cannot know if a detected energy signature is an allied or enemy bug. The bug that is about to move signals that it is about to move and it will signal again when it is done moving. When friendly bugs die, they send a signal that they are dying. This allows synchronization of the data structures if a death event is detected between "about to move" and "done moving" events[14].

The "HQ" is used to coordinate the group data structures and abstract employment of the data structures. Knowing the (relative) positions of all friendly bugs allows key pieces of information or capabilities:

- If surrounded by allies, the direction(s) to the closest edge of the colony can be determined;
- Environmental sensing is minimized, by not sensing known ally positions;
- Search scores can be maintained (see Section B.3.2);
- Detailed statistics of enemy encounters can be maintained; and
- Cloaking can be used, since the positions of cloaked allies will be known.

B.3.8 Scan Caching

Agent variants in the exploration phase should have multiple cilia to maximize movement and exploration of the environment. Since each move is only one cell away, agents will often have to re-consider positions that they previously knew about. On a single turn for an agent, the results of scans can be cached to prevent energy waste by re-scanning positions where the data cannot have changed.

Attempting to cache scanning information between agents is invalid, partially because of the unknown turn order and partially because other agents may move more than one cell.

Table B.2 outlines the energy costs for the worst-case energy costs. The table only covers up to eight moves between this is the maximum reasonable number of organ positions available for cilia. One position is normally required for a sensor and one for a budder. The worst case occurs when each move is along the same diagonal direction. The initial position is exposed to eight cells, and each subsequent move exposes five new cells. Scanning a position assumes an energy scan.

[14] There are special cases such as the mover dying due to defensive damage. They are accounted for in the design but are excluded here for brevity.

Table B.2: Benefits of Scan-Caching (Worst-Case)

Moves	Basic Scanning Cost	Cached Scanning Cost	Movement Cost	Total Basic Cost	Total Cached Cost
1	16	16	20	32	32 (100%)
2	32	21	40	72	61 (85%)
3	48	26	60	108	86 (80%)
4	64	31	80	144	111 (77%)
5	80	36	100	180	136 (76%)
6	96	41	120	216	161 (75%)
7	112	46	140	252	186 (74%)
8	128	51	160	288	211 (74%)

B.3.9 Defeating Cloaking

Although expensive to execute, cloaking can be mitigated. If an agent suspects cloaking is being used, they could spawn minimum energy bugs ("recce bugs") onto apparently open ground. If the bug survives then the ground actually is open, but if it dies then there is a cloaked bug there. Since no information about the energy levels of cloaked bugs can be obtained, methods to deal with the detected bug are explored in the agent designs.

Since each recce bug costs 101 energy to employ (100 for the budder use, 1 to be placed in the environment), the worst time to employ this technique is in the exploration phase since the energy could be used more effectively to explore the environment. This technique should only be employed when there is a high probability of cloaked enemy present.

Due to the turn ordering of the simulation, adjacent allied bugs will not need to spawn recce bugs on the same cells. The recce bugs will survive (unless killed directly) until the next simulation turn. Having the recce bugs killed by enemies is actually a fringe benefit because it would waste their movement or energy for unnecessary spawning.

B.3.10 Defensive Organs

Table B.1 presented the organs and their relative cost. The only two organs that provide defensive damage are spikes and poison (created with the poison gland). When using the kamikaze strike (see Section B.3.6), the bug spawns a new child that has only defensive organs added in its initialization phase, which occurs before cost. This leads to the question of which organs will do the most damage. While a decision about whether or not to use the kamikaze strike depends on the strength of the parent bug and the target, the efficiency or which organs to use is fixed. The base cost to spawn the attacker is 100,

which is the cost to use the budder[15]. This base cost will be ignored for the rest of this section, to avoid confusion when considering the organs that are added. Spikes cause 200 damage and are added at a fixed rate of 100 energy each. Poison glands cost 500 to add, but then each poison unit (up to a maximum of 1000 units per poison gland) only cost 1 energy. Each poison unit causes 4 damage. If it is assumed that a bug would create all 10 organs as spikes, the total cost would be 1000 energy for 2000 damage. Using this same amount of energy for poison, the bug could create a poison gland with 500 poison units, which is also capable of 2000 damage. If another organ could be added then poison would be more effective after this equilibrium. When between 1000-4000 is required, a single poison gland with sufficient poison is the most effective option. Past 4000 damage requires additional organs, at which point the high cost of the poison gland makes spikes more effective until the maximum organ count is reached.

The conclusion of the above comparison is that spikes are always more efficient as long as there is sufficient space in the organ limit. To deal extremely high amounts of damage, a combination of poison glands and spikes are required. Dealing less than 200 damage is never efficient, due to the base cost of spawning and the minimum of one spike organ. The maximum amount of damage that can be dealt with a single bug is 40000, generated by an attacker with 10 fully loaded poison glands. This scenario would cost the parent bug 15100 energy, although it is extremely unlikely. A magnified perspective of the damage function is given in Figure B.5, while the total damage function is given in Figure B.6.

[15] For clarity, the "attacker" is the child bug created by the parent bug. The parent bug is the active bug that selects the enemy target.

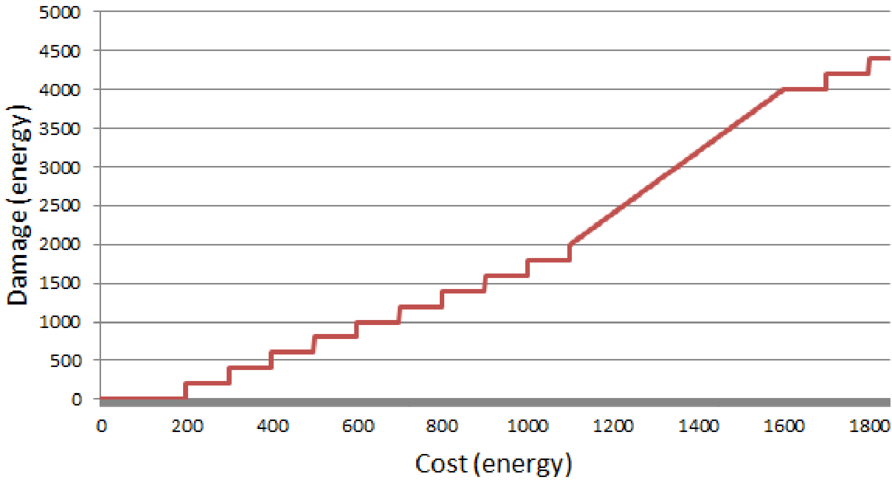


Figure B.5: Magnification of the Attack Bug Damage Function

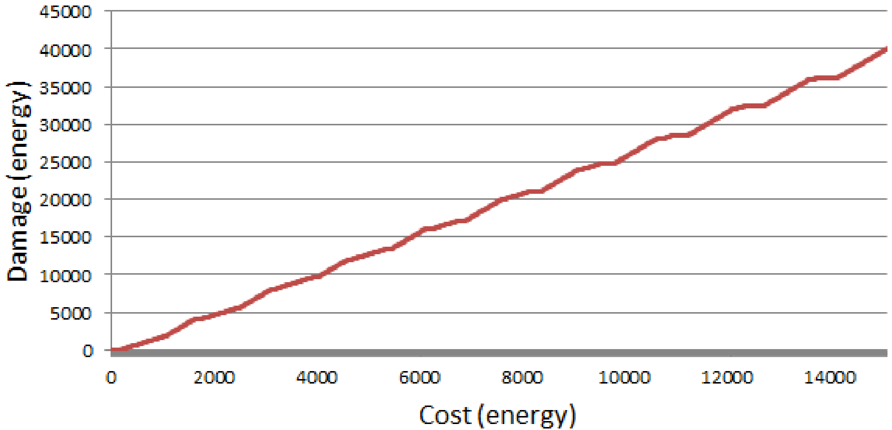
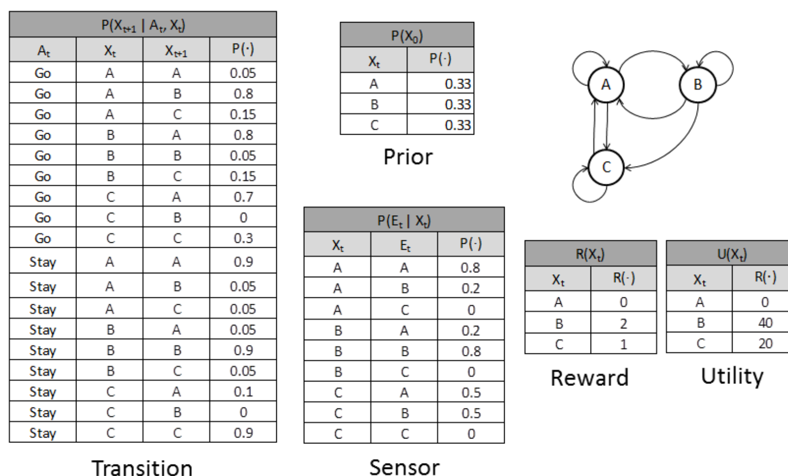


Figure B.6: Attack Bug Damage Function

C Particle Filtering and Decision Trees

This Annex presents a detailed example for particle filtering to illustrate how the algorithm works. Particle filtering is discussed at the end of Section 2.3.5. The system used in this example is the toy system as presented in Section 4.2.1, which is repeated in Figure C.1 to facilitate reading probabilities from the various sub-models. After the particle filtering example is worked through, the example is extended to build a full decision tree of depth two to illustrate how to solve a decision tree. A population size of ten is used throughout for the particle filtering algorithm, which is quite small for an actual application.



Transition

$P(X_{t+1} \mid A_t, X_t)$			
A_t	X_t	X_{t+1}	$P(\cdot)$
Go	A	A	0.05
Go	A	B	0.8
Go	A	C	0.15
Go	B	A	0.8
Go	B	B	0.05
Go	B	C	0.15
Go	C	A	0.7
Go	C	B	0
Go	C	C	0.3
Stay	A	A	0.9
Stay	A	B	0.05
Stay	A	C	0.05
Stay	B	A	0.05
Stay	B	B	0.9
Stay	B	C	0.05
Stay	C	A	0.1
Stay	C	B	0
Stay	C	C	0.9

Prior

$P(X_0)$	
X_t	$P(\cdot)$
A	0.33
B	0.33
C	0.33

Sensor

$P(E_t \mid X_t)$		
X_t	E_t	$P(\cdot)$
A	A	0.8
A	B	0.2
A	C	0
B	A	0.2
B	B	0.8
B	C	0
C	A	0.5
C	B	0.5
C	C	0

Reward

$R(X_t)$	
X_t	$R(\cdot)$
A	0
B	2
C	1

Utility

$U(X_t)$	
X_t	$R(\cdot)$
A	0
B	40
C	20

Figure C.1: Toy 3-State Model

C.1 Particle Filtering

Bayes' rule is crucial to the particle filtering algorithm and is given below. It can be found in most introductory probability textbooks, [16] is one example. The form of the equation presented has the set of all possible evidence E and the set of all possible states X. The t subscripts denote time. Membership in a set is denoted by a lower-case letter, e.g. $x \epsilon X$ means a single possible state as opposed to X which is all states.

$$P(X_t|e_t) = \frac{P(e_t|X_t)P(X_t)}{P(e_t)} = \frac{P(e_t|X_t)P(X_t)}{\sum_{x \epsilon X} P(e_t|x_t)P(x_t)} \tag{C.1}$$

Assume that a state is sampled from the prior distribution. Our initial state will be A in this example. To begin the particle filtering algorithm, we obtain an initial sample. The distribution that the initial sample is drawn from depends when sampling occurs. Since this example is just starting, sampling from the prior distribution is ideal. If attempting to sample in the middle of a simulation, an estimate of the current distribution will be required. The initial sample for this example will be:

A	B	C
3	4	3

Note that main benefit of using particle filtering is that the quantity of samples of a state can be used as an estimate for the probability of that state. For example, $P(A) = \frac{N(A)}{N(A)+N(B)+N(C)} = 0.3$.
Evidence is generated from the current state. For this example, $e_0 = A$. For each $x \epsilon X$ we need to calculate:

$$P(X_0|e_0 = A) = \frac{P(e_0 = A|X_0)P(X_0)}{\sum_{x \epsilon X} P(e_0 = A|x_0)P(x_0)}$$

Since the denominator will be common for all state elements, it will save space

to calculate that first:

$$\begin{aligned}
\sum_{x \epsilon X} P(e_0 = A|x_0)P(x_0) &= P(e_0 = A|x_0 = A)P(x_0 = A) \\
&\quad + P(e_0 = A|x_0 = B)P(x_0 = B) \\
&\quad + P(e_0 = A|x_0 = C)P(x_0 = C) \\
&= (0.8)(0.3) + (0.2)(0.4) + (0.5)(0.3) \\
&= 0.47
\end{aligned}$$

Performing the calculation for each $x \epsilon X$ yields:

$$\begin{aligned}
P(x_0 = A|e_0 = A) &= \frac{P(e_0 = A|x_0 = A)P(x_0 = A)}{0.47} \\
&= \frac{(0.8)(0.3)}{0.47} \\
&= \frac{0.24}{0.47} \\
&= 0.51
\end{aligned}$$

$$\begin{aligned}
P(x_0 = B|e_0 = A) &= \frac{P(e_0 = A|x_0 = B)P(x_0 = B)}{0.47} \\
&= \frac{(0.2)(0.4)}{0.47} \\
&= \frac{0.08}{0.47} \\
&= 0.17
\end{aligned}$$

$$\begin{aligned}
P(x_0 = C|e_0 = A) &= \frac{P(e_0 = A|x_0 = C)P(x_0 = C)}{0.47} \\
&= \frac{(0.5)(0.3)}{0.47} \\
&= \frac{0.15}{0.47} \\
&= 0.32
\end{aligned}$$

This provides the probability vector $\langle 0.51, 0.17, 0.32 \rangle$, which is re-sampled to obtain an estimate of the actual state given then current evidence. One possible sampling is:
This completes the filtering algorithm. The process is summarized in Figure C.2.

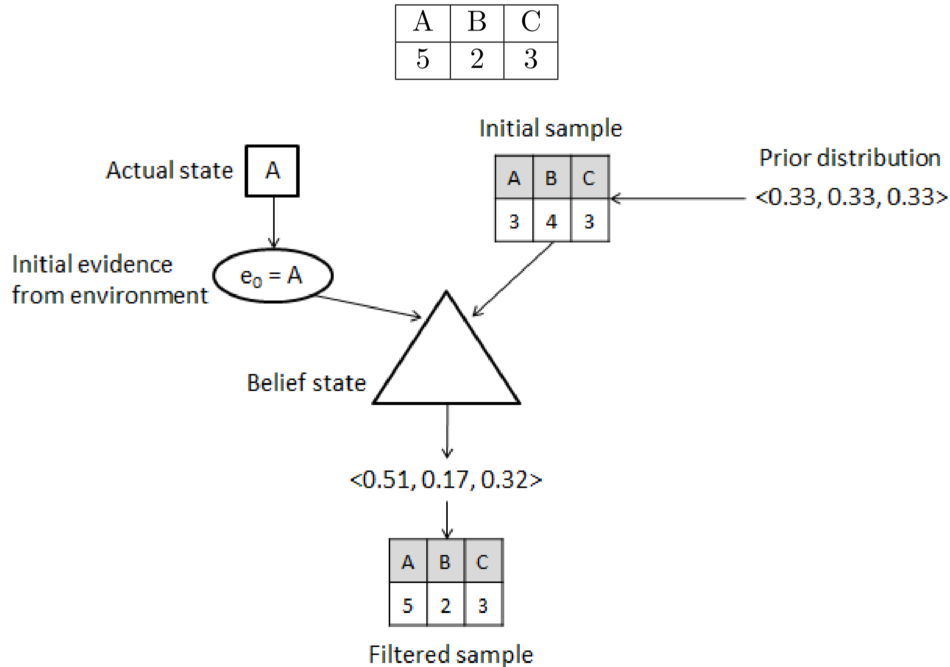


Figure C.2: Particle Filtering Algorithm

C.2 Decision Trees

Generally a decision tree is developed by examining likely outcomes for different decisions by projecting forward. At each decision point, each potential action is examined and the relative reward is compared to the other actions. When multiple steps are considered the rewards of the future actions are factored into the assessment of which decision is best by averaging the reward against the likelihood of the future state. At each decision point, the decision that is chosen is one of the decisions with the highest overall reward. Decision trees are discussed in [20].

Creation of a decision tree for a POMDP follows this same process. The difference between a fully and partially observable case is that the decision points are belief distributions instead of state distributions. To simplify the presentation, only belief distributions will be considered and as they will be presented graphically later they will be called belief nodes. Propagating the decisions forward involves obtaining a sample from the belief node and apply-

ing the transition model for each sample, which results in a "chance node". Evidence can then be generated for the transitioned sample with a relative weight leading to the next decision (belief) node. Note that while samples are used to build a branch for a particular action, this is not particle filtering.

As an example, the decision tree begun in Section C.1 will be propagated to two decisions and then the decision tree will be solved. Solving the decision tree involves averaging the children of chance nodes and selecting the child with the maximum reward from belief nodes.

To aid in mapping the calculations from the text to the resulting figure, the notation B_t and C_{ta} will be used. B denotes a belief node while C denotes a chance node. The t and ta subscripts denote the turn, evidence and action sequence that led to the node. For example, B_0 is the first belief node and C_{0-GO} is the chance node that results from propagating forward the GO action.
Recall the post-filtering sample obtained from B_0 was given as:

A	B	C
5	2	3

Recall that for $a_t = GO$ the extract from the transition model is:

x_0	x_1		
	A	B	C
A	0.05	0.80	0.15
B	0.80	0.05	0.15
C	0.7	0.0	0.3

Which results in a possible sampling for $a_0 = GO$ of:
Similarly the same process for $a_0 = STAY$ potentially yields:
To calculate the reward at C_{0-GO}, we multiply the probability of the state by the corresponding entry in the reward function:

$$\begin{aligned} R(C_{0-GO}) &= P(x' = A)(R(A)) + P(x' = B)(R(B)) + P(x' = C)(R(C)) \\ &= (0.4)(0.0) + (0.4)(2) + (0.2)(1) \\ &= 1.0 \end{aligned} \tag{C.2}$$

x_0	x_1		
	A	B	C
A	0	4	1
B	2	0	0
C	2	0	1
	4	4	2

A	B	C
5	2	3

And for C_{0-STAY}:

$$\begin{aligned} R(C_{0-STAY}) &= P(x' = A)(R(A)) + P(x' = B)(R(B)) + P(x' = C)(R(C)) \\ &= (0.5)(0.0) + (0.2)(2) + (0.3)(1) \\ &= 0.7 \end{aligned}$$

Next, the evidence generated by C_{0-GO} is required. To obtain $P(E_1)$, we apply the sensor model and normalize the result. Note that since $P(E_t = C) = 0$ is always true, we omit considering C whenever evidence is discussed.

$$\begin{aligned} P(e_1 = A) = &P(e_1 = A|x_1 = A)P(x_1 = A) + P(e_1 = A|x_1 = B)P(x_1 = B) \\ &+ P(e_1 = A|x_1 = C)P(x_1 = C) \\ = &(0.8)(0.4) + (0.2)(0.4) + (0.5)(0.2) \\ = &0.5 \end{aligned}$$

$$\begin{aligned} P(e_1 = B) = &P(e_1 = B|x_1 = A)P(x_1 = A) + P(e_1 = B|x_1 = B)P(x_1 = B) \\ &+ P(e_1 = B|x_1 = C)P(x_1 = C) \\ = &(0.2)(0.4) + (0.8)(0.4) + (0.5)(0.2) \\ = &0.5 \end{aligned}$$

Then evidence is generated for C_{0-STAY}:

$$
\begin{aligned}
P(e_1 = A) &= P(e_1 = A|x_1 = A)P(x_1 = A) + P(e_1 = A|x_1 = B)P(x_1 = B) \\
&\quad + P(e_1 = A|x_1 = C)P(x_1 = C) \\
&= (0.8)(0.5) + (0.2)(0.2) + (0.5)(0.3) \\
&= 0.59
\end{aligned}
$$

$$
\begin{aligned}
P(e_1 = B) &= P(e_1 = B|x_1 = A)P(x_1 = A) + P(e_1 = B|x_1 = B)P(x_1 = B) \\
&\quad + P(e_1 = B|x_1 = C)P(x_1 = C) \\
&= (0.2)(0.5) + (0.8)(0.2) + (0.5)(0.3) \\
&= 0.41
\end{aligned}
$$

Next a belief node is generated for each evidence branch by assuming the evidence is fixed and applying particle filtering. This follows the same process as Section C.1, so for brevity the calculations are omitted. At this point the partial decision tree is given in Figure C.3.

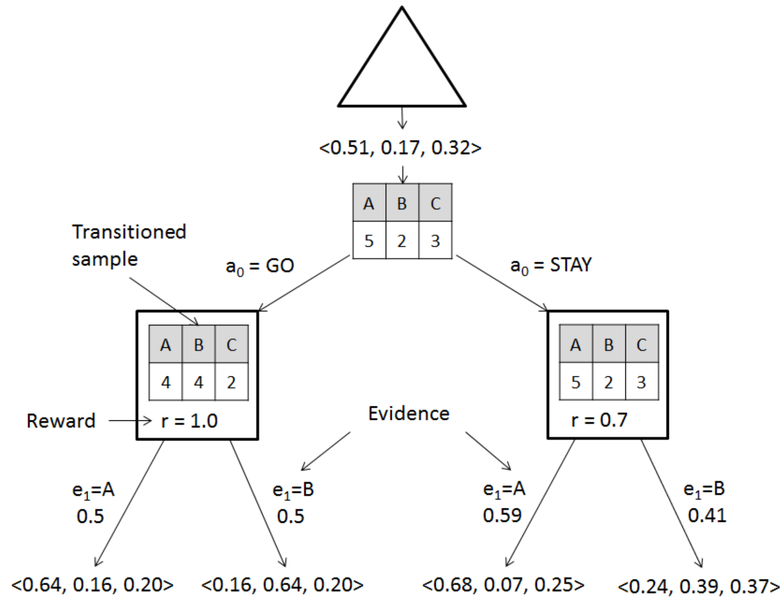


Figure C.3: Partial Decision Tree - Once Evidence is Generated

The rest of the decision tree is built by completing these same steps for each successive layer of the tree. The process changes when the desired depth is

reached. At this point an estimate of the long-term utility of the leaf belief nodes is calculated used the utility function, which is by necessity a heuristic. The utility for the belief node is calculated the same way as Equation (C.2), except the utility function is used instead of the reward function.
Figure C.4 shows the completed tree with all samples and distributions. Figure C.5 is the same tree, but abstracted to show only the utilities, rewards and evidence distributions. Finally Figure C.6 presents the solved tree, which shows that the preferred action is $a_0 = GO$.

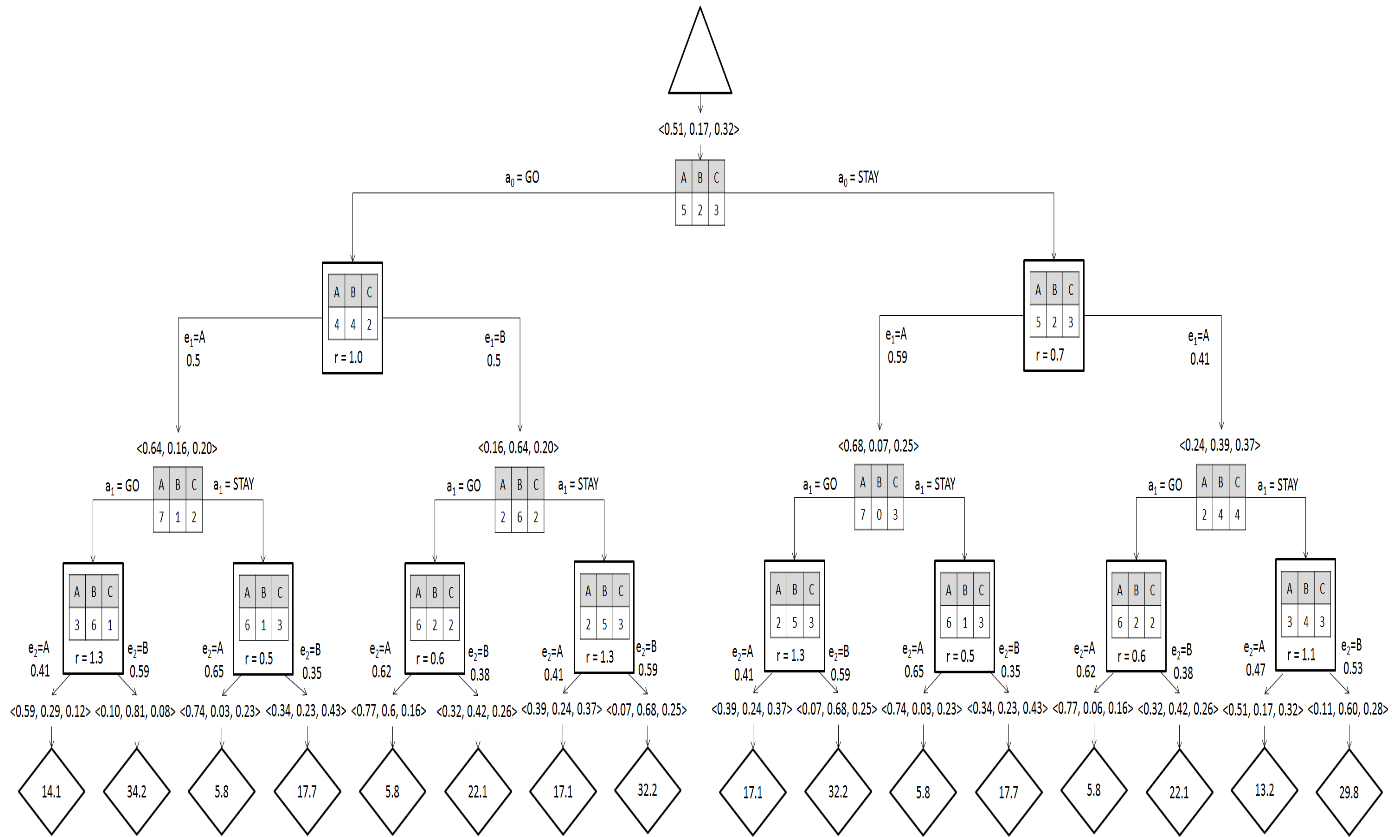


Figure C.4: Full Decision Tree - with Details

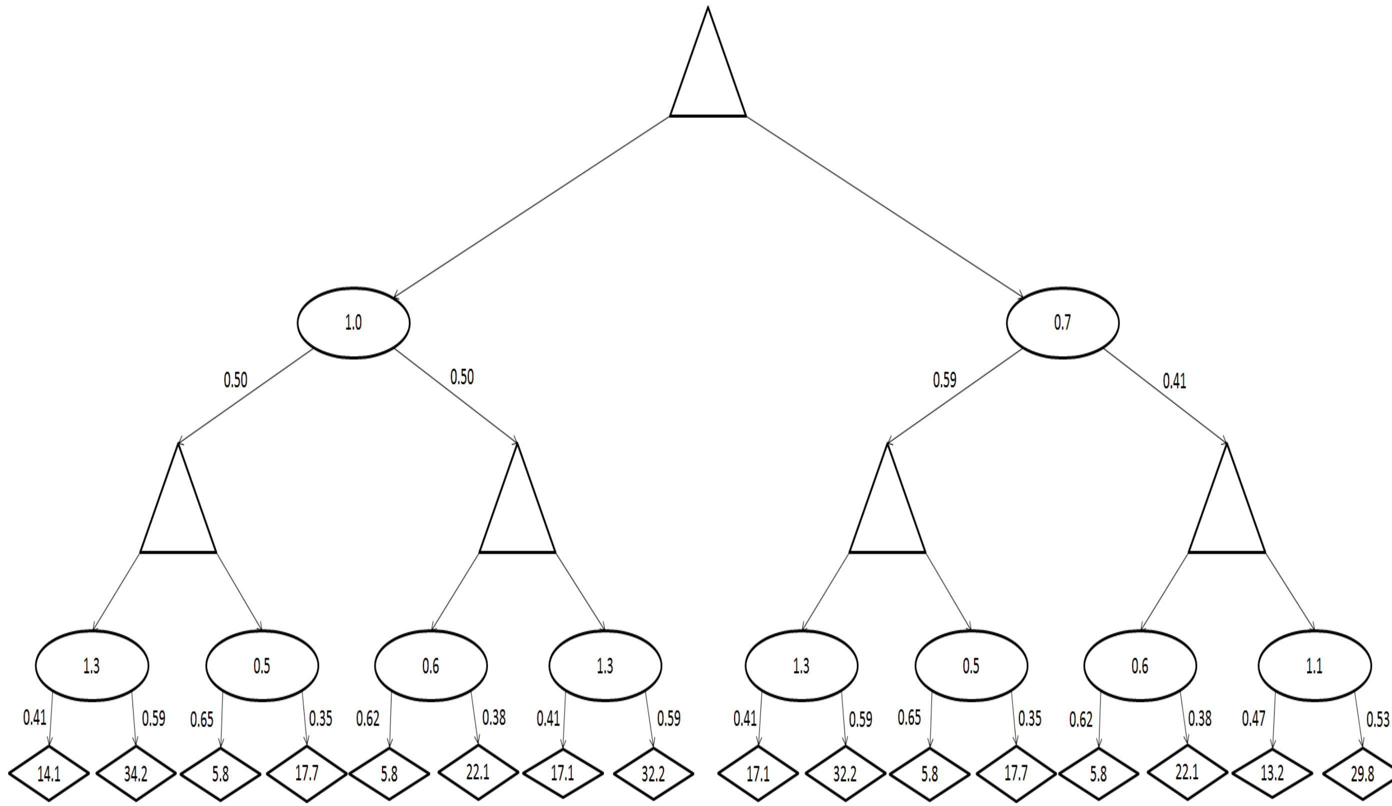


Figure C.5: Full Decision Tree - Abstracted

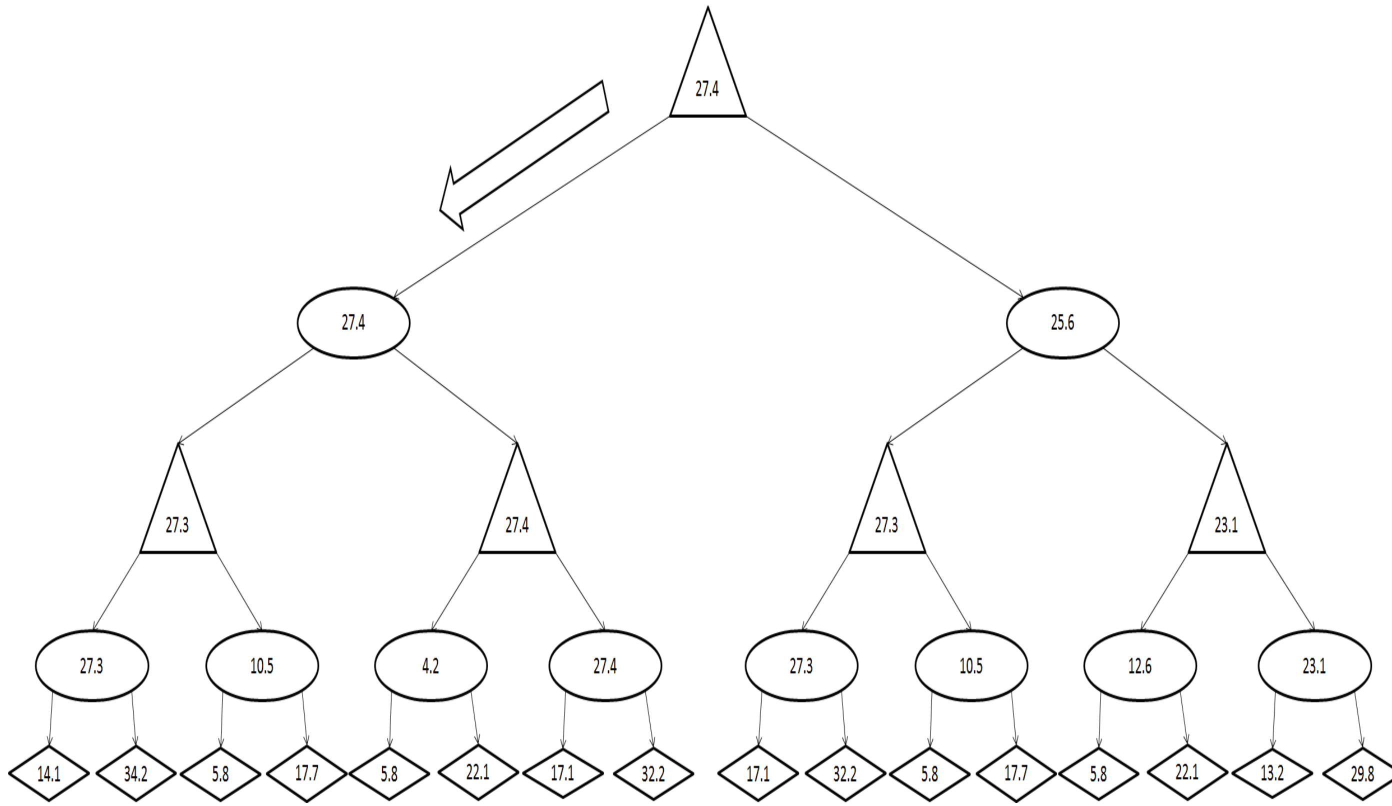


Figure C.6: Full Decision Tree - Solved

D Supplementary Data

The contents of the data DVD are as follows:

- Source.zip - The simulation source code, packaged as an Eclipse project.
- Website.zip - The source code for the experiment website[1].
- bugbattleai.sql - The raw data from the simulation trials.
- Graphs.zip - The Excel files that graphs were generated from.

[1] This does not contain server configuration information, it is only the front-end code.

Bibliography

[1] E. Alpaydin. *Introduction to Machine Learning.* The MIT Press, 1st edition, 2004.

[2] C. Amato and F.A. Oliehoek. Scalable planning and learning for multiagent pomdps. *CoRR*, abs/1404.1140, 2014.

[3] R. Bellman. *Dynamic Programming.* Princeton University Press, Princeton, NJ, USA, 1 edition, 1957.

[4] A. Borshchev and A. Filippov. From system dynamics and discrete event to practical agent based modeling: Reasons, techniques, tools. In *The 22nd International Conference of the System Dynamics Society.* System Dynamics Society, July 2004.

[5] L. Champagne et al. Proceedings of the 2003 winter simulation conference. In S. Chick et al., editors, *Search Theory, Agent-Based Simulation, and U-Boats in the Bay of Biscay*, volume 1, pages 991–998, 2003.

[6] M. Dorigo and G. Di Caro. The ant colony optimization metaheuristic. In David Corne et al., editors, *New Ideas in Optimization*, pages 11–32. McGraw-Hill Ltd, 1999.

[7] M. Dorigo et al. Ant colony optimization. *IEEE Computational Intelligence Magazine*, 1(4):28–39, Nov 2006.

[8] A. Doucet and A.M. Johansen. A tutorial on particle filtering and smoothing: Fifteen years later. *Handbook of nonlinear filtering*, 12(656-704):3, 2009.

[9] J. Ferber. *Multi-Agent System: An Introduction to Distributed Artificial Intelligence.* Harlow: Addison Wesley Longman, 1999.

[10] H. Fujita and S. Ishii. Model-based reinforcement learning for partially observable games with sampling-based state estimation. *Neural Computation*, 19(11):3051–3087, 2007.

[11] T. Hester and P. Stone. Learning and using models. In Marco Wiering and Martijn van Otterlo, editors, *Reinforcement Learning: State of the Art*, pages 111–141. Springer, 2012.

[12] C.M. Macal and M.J. North. Agent-based modeling and simulation: Desktop abms. In S.G. Henderson et al., editors, *Proceedings of the 2007 Winter Simulation Conference*, pages 95–106, 2007.

[13] N. Metropolis and S. Ulam. The monte carlo method. *Journal of the American Statistical Association*, 44(247):335–341, September 1949.

[14] K. Murphy. *Dynamic Bayesian Networks: Representation, Inference and Learning*. PhD thesis, UC Berkley, 2002.

[15] E. Parzen. *Stochastic Processes*. Holden-Day Series in Probability and Statistics. Holden-Day, Inc., 1965.

[16] P. Pfeiffer. *Concepts of Probability Theory*. Dover Publications, Inc., 2nd edition, 1978.

[17] D.L. Poole and A.K. Mackworth. *Artificial Intelligence: Foundations of Computational Agents*. Cambridge University Press, 2010.

[18] B. Rathbun and S. Hill. Research ethics board - preliminary work. Personal communication, April 2016. Confirmation by Dr Hill that a Research Ethics Board submission would not be required for a proposed distributed computing environment.

[19] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1st edition, 1995.

[20] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, 3rd edition, 2009.

[21] S.M. Sanchez and T.W. Lucas. Proceedings of the 2002 winter simulation conference. In E. Yucesan et al., editors, *Exploring the World of Agent-Based Simulations: Simple Models, Complex Analyses*, volume 1, pages 116–126, 2002.

[22] N. Schieritz and P.M. Milling. Modeling the forest or modeling the trees: A comparison of system dynamics and agent-based simulation. In R.L Eberlein et al., editors, *Proceedings of the 21st International Conference*. System Dynamics Society, The publisher, 2003.

[23] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 1(60):51–92, 1993.

[24] D. Silver and J. Veness. Monte-carlo planning in large pomdps. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 2164–2172. Curran Associates, Inc., 2010.

[25] M.T.J. Spaan. Partially observable Markov decision processes. In M. Wiering and M. van Otterlo, editors, *Reinforcement Learning: State of the Art*, pages 387–414. Springer Verlag, 2012.

[26] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*, volume 1. MIT press Cambridge, 1998.

[27] G. Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[28] P. Tozour. Influence mapping. In Mark DeLoura, editor, *Game Programming Gems 2*, pages 287–297. Charles River Media, 2001.

[29] W. Turkett Jr and J. Rose. Planning with agents: An efficient approach using hierarchical dynamic decision networks. In *Proceedings of the Fourth International Workshop on Engineering Societies in the Agent World, London, England*, 2003.

[30] M. van Otterlo and M. Wiering. *Reinforcement Learning and Markov Decision Processes*, pages 3–42. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[31] C.J.C.H. Watkins and P. Dayan. Technical note: Q-learning. *Machine Learning*, 8(3):279–292, 1992.

[32] D. Weyns et al. Environments for multiagent systems state-of-the-art and research challenges. In H. Van Dyke Parunak D. Weyns, editor, *Environments for Multi-Agent Systems*, pages 1–47. Springer, 2004.

[33] D. Weyns et al. Environments for situated multi-agent systems: Beyond infrastructure. In F. Michel D. Weyns, H. Van Dyke Parunak, editor, *Environments for Multi-Agent Systems II*, pages 1–17. Springer, 2005.

[34] D. Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

[35] N. Wirth and M. Gallagher. An influence map model for playing ms. pac-man. In *2008 IEEE Symposium On Computational Intelligence and Games*, pages 228–233, Dec 2008.

[36] L. Yuefeng and Z. An. Multi-agent system and its application in combat simulation. In *Computational Intelligence and Design, 2008. ISCID '08. International Symposium on*, volume 1, pages 448–452, Oct 2008.