Lehigh University
Lehigh Preserve

Theses and Dissertations

2013

Stochastic Explanations: Learning From Mistakes In Stochastic Domains

Giulio Finestrali
Lehigh University

Follow this and additional works at: http://preserve.lehigh.edu/etd

Part of the Computer Sciences Commons

Recommended Citation

STOCHASTIC EXPLANATIONS: LEARNING FROM MISTAKES IN STOCHASTIC DOMAINS

by

Giulio Finestrali

A Thesis

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

In

Computer Science

Lehigh University

Graduating Class of May 2013

This thesis is accepted and approved in partial fulfillment of the requirements for the Master of Science.

_____

Date

_____

Thesis Advisor: Prof. Héctor Muñoz-Avila

_____

Chairperson of Department: Prof. Daniel P. Lopresti

Acknowledgements

Table of Contents

List of Figures

Abstract

Explaining anomalies in a stochastic environment is a complex task and a very active research field. In this thesis we present our contributions to this fascinating topic by introducing the concept of Stochastic Explanations, which associate to any unexpected event or anomaly a probability distribution over the possible causes of the anomaly. In this thesis, we present the EXP_GEN agent, which uses stochastic explanations to explain anomalies, implementing a synergy of Case-Based Reasoning and Reinforcement Learning mechanisms to, respectively, store and retrieve as cases *(event, stochastic explanation)* pairs, and to learn the probability distribution of the stochastic explanation for each anomaly. We claim that an agent using stochastic explanations will react faster to unexpected events than an agent that uses a deterministic approach to explain anomalies. We compare the performance of EXP_GEN against an agent which utilizes a greedy heuristic to explain anomalies.

Unexpected events should be considered as a huge opportunity for an intelligent agent to learn something new about the environment. Providing explanations to these events is just the first step: we must reutilize the previous experience and explanations to avoid making the same mistakes in the future. For this purpose, we present the GENERATE_GOAL+ algorithm, which replaces the basic goal generation mechanism of the EXP_GEN agent. This algorithm takes into consideration the previous mistakes made by the agent and uses them to generate better goals. In this way, the agent is effectively *learning from mistakes*, improving its

performance over time. We consider the ability to learn from one's own mistakes crucial to the implementation of more complex intelligent agents in the future. In our results, we show how the EXP_GEN agent that uses GENERATE_GOAL+ greatly outperforms the same agent that uses naïve goal generation.

To show the effectiveness of learning from mistakes, we also present an enhanced version of the GENERATE_GOAL+ algorithm, called CombatPlan. Here we show how the EXP_GEN agent can handle broader and more complex scenarios and learn effectively by the mistakes it makes. To show this, we train the agent for a certain number of iterations, so that it makes mistakes and explains them. We then show how the performance of the agent on the test scenario is directly related to the amount of training performed, or in other words, to the amount of explanations generated during the training phase.

Chapter 1   Introduction

Artificial Intelligence (AI from here on) is a multifaceted field whose objectives and methods are very often misinterpreted and not always intuitive to the general public. While today's society is already accustomed to seeing a computer outperform a human being in many activities, we are still baffled when we witness a machine outperforming a person in a task that we consider *human*. For example, we are not particularly surprised seeing a computer executing billions of calculations per second; but we were astonished when, in February 2011, IBM's supercomputer *Watson* played a game of the popular quiz Jeopardy against two of the best human players in the world and won.

Why are we surprised (and maybe intimidated) by a machine like Watson, and not at all by our personal computers? When a machine like Watson answers correctly to the very complex questions presented to it during the quiz, what impresses us more than the accuracy of the answers is that we identify in the machine characteristics that we attribute to human beings. The cognitive process required to understand the question and retrieve the answer is something we are not used to seeing in a computer, but it is something we perform on a regular basis as human beings.

The ultimate goal of AI is studying and eliminating the gap that exists between our interpretation of the world that surrounds us and the computers' interpretation. Such a goal is extremely challenging, especially considering how

human beings change their interpretation of the environment continuously through several mechanisms.

From the day we are born until the day we die, our brain is constantly at work, acquiring new information and combining it with the knowledge we already have, thus enriching our description of the world. Traditional computer programs operate in a dramatically different way, because they lack a mechanism that allows them to create novel content from the input data that is available. AI aims to fill this void by enriching traditional computing by enabling software to simulate human-like learning processes, thus increasing the software performances with time.

1.1 Context

As previously mentioned, humans modify their current knowledge of the world in many ways. For example, a basic and very effective way is learning from *feedback*. While this mechanism happens throughout the entire life of an individual, it is particularly effective in the early stages of life; in fact, when babies have not yet developed communication skills, they learn mainly through the feedback that the environment provides to them. This feedback can come in different ways, it can be physical or emotional, but in the end what matters is that the individual performs an action and he is able to *classify* the outcome of such action by observing the feedback provided by the environment as a result of his action. When the feedback is positive the action is "good" and can be reiterated in the future; if the feedback is negative, then the action should not be repeated in the future in order to avoid receiving another negative feedback. This is the mechanism that each and every one of us used

to learn that you can get burnt by touching a flame, and that it is unwise to eat directly from a steaming bowl of soup. Reinforcement Learning (RL) is a software implementation of the feedback learning mechanism and will be discussed more in depth in the next chapter.

Another very common way of learning is utilizing past experiences to categorize new situations when they occur. This is a very natural process: whenever a new problem happens, we try to find a similar situation that might have occurred in the past. If we can recall a similar problem that we previously solved, we might be able to apply the same solution, or a modified version of it, to the new problem. This method of learning takes the name of Case-Based Reasoning (CBR) and it will also be discussed in the next chapter.

We briefly introduced two of the main learning mechanisms that we believe human beings (and intelligent agents) commonly use to increase their knowledge of the environment. A third mechanism, which will represent the fulcrum of this thesis, is *learning from mistakes*. This mechanism relies on the following assumption: recognizing a mistake is a huge learning opportunity. To take advantage of such opportunity, we must be able to *explain* why the mistake was made, what caused it, and how to avoid it in the future. Notice how this is a huge step forward in respect to the mere feedback system we previously discussed. If before we were satisfied with just collecting the feedback from the environment, now we want to be able to explain why this feedback was generated. This is a necessary step towards a more complete understanding of the environment. Once we know the reason of the

feedback, we are then able to reason more accurately on the real state of the environment, and how to avoid committing the same mistake in the future. We will discuss in Chapter 4 how this mechanism is fundamental in case the nature of the environment is stochastic and not strictly deterministic. In such domains, mistakes can also be in the form of *unexpected events*. In this case the mistake is not being able to foresee the occurrence of such events. We refer to mistakes in general with the term *discrepancy;* a thorough and more formal definition of this term will be given in Chapter 4.

Explaining discrepancies in a stochastic domain is a non-trivial task, which can be subdivided in two tasks: given a discrepancy, provide a set of possible explanations for it; given the set of possible explanations, select the correct one from it. In this thesis, we concentrate on performing the second task efficiently. The first task represents a very active and extremely intriguing research topic, and it is left for future work. We will cover more of this in Chapter 7.

1.2 Contributions

This thesis presents a novel approach to selecting explanations for discrepancies in a stochastic domain by using a synergy of CBR and RL. We introduce the concept of Stochastic Explanation, which particularly fits the uncertainty and dynamicity of stochastic environments. We present EXP_TEAM, an intelligent agent that uses Stochastic Explanations to react to unexpected events in a Real-Time Strategy game, and EXP_TEAM+, an extension of EXP_TEAM which learns from previously explained discrepancies to increase the agent's performances. Finally, we introduce

CombatPlan, an agent that utilizes EXP_TEAM in its training phase, and then uses the learned explanations in more complex scenarios within the same RTS environment.

1.3 Organization

In Chapter 2 we set the foundations for the rest of the thesis: we explain the CBR and RL mechanisms in depth, and how we can use them together to improve an intelligent agent's learning performance. Chapter 3 contains an analysis of the Related Work on explanations, especially regarding RTS domains. Chapter 4 explains in depth the mechanisms of Stochastic Explanations and how they interact with the CBR and RL systems, also showing the pseudo-code of the EXP_TEAM agent. Chapter 5 goes into the implementation details of the EXP_TEAM agent, presenting the most important algorithms used by the agent. This will be at a lower level than the pseudo-code presented in the previous Chapter. Chapter 6 presents the empirical study for the three agents: a thorough presentation of the tests that were run and the respective results. Finally, Chapter 7 contains the Conclusions of the thesis, summing up the concepts presented and the results achieved, presenting ideas for Future Work on this topic.

Chapter 2   Preliminaries

In this Chapter we will introduce some key concepts which are fundamental for the rest of this thesis. First, we will discuss Reinforcement Learning, a feedback-based learning framework, showing its characteristics, strengths, and also its drawbacks. Then, we will discuss another learning mechanism: Case-Based Reasoning. These two learning paradigms are fundamental to understand Explanations, and in particular, Stochastic Explanations (Chapter 4).

2.1 Reinforcement Learning

RL is a feedback-based, or *reward*-based, learning paradigm (Sutton & Barto, 1998). In a RL system, an intelligent agent starts its execution knowing very little, if anything, about the environment it operates in. The agent can interact with the environment by performing *actions*. When it does so, the agent receives a feedback from the environment, which takes the more technical name of *reward*. This reward is a numerical entity which the agent uses to learn about the effects that its own actions have on the environment. Positive rewards identify a state of the environment that is "better" than the previous one for the agent. The definition of *better* in this case is context-dependent, and varies from an application to another. A negative reward, instead, identifies a state of the environment which is "worse" than the previous one for the agent.

A RL agent is programmed to maximize the rewards he obtains from the environment, but since at the beginning of its execution the agent does not know

which effects each action will have on the environment, the performances of a RL system during its early executions are usually poor. RL is a learning paradigm based on repetition, it is characterized by a slow-climbing learning curve, and very often achieves great efficacy if provided with a sufficient amount of training iterations.

While RL is very effective, there are some caveats to keep in mind when using it. First, RL does not provide the agent with an in-depth description of the environment. The rewards are numeric values which do provide the agent with a generic idea of what is beneficial and what is not; from such values an agent can learn what is the best action to take at a given moment. Nonetheless, the agent is following a reward, not a line of thought. It learns that performing a certain action in a given state provides a good reward, and therefore performs the action, without caring what are the implications of doing so, or, in other words, why the reward is positive for such action. Often this is good enough, but if we want to implement more complex reasoning frameworks, then we need to expand the RL paradigm.

Another caveat when using RL is that it is usually slow. When the environment presents many elements or can be interacted with in many ways, the number of action-reward combinations that an agent has to learn is very high. Also, in many domains the reward for an action in a given state is influenced by the actions taken previously; in this case the number of training rounds required for RL to perform adequately can be too high to be acceptable.

Finally, and most importantly for the sake of this thesis, RL in its original form has huge performance issues when utilized in stochastic domains. The problem

here is that an agent using RL needs to know the rewards for each possible action it can take when it finds itself in a certain state. This reward is something that the agent learned throughout previous iterations, when it found itself in the same state and it performed some action, collecting and memorizing the reward from the environment. We have this kind of associations:

$$< state, action > = reward$$

In a deterministic domain, each action will always provide the same reward every time it is performed in a certain state, and this characteristic lets the agent learn which action to perform given the current state. In a stochastic domain this assumption fails, because the pair (state, action) does not yield the same reward every time. This is an important characteristic of stochastic domains, and requires RL to either perform much more training, or it makes it unusable in case of extremely unpredictable domains where the rewards can switch from positive to negative very quickly. Stochastic Explanations rely on a synergy of RL and CBR: we introduce this concept at the end of this Chapter, but we will expand it in much more detail in Chapter 4.

2.2 Case-Based Reasoning

The CBR framework relies on the very natural intuition that past experiences can help us solve new problems. There exist many CBR implementations (Wilke, Lenz, & Wess, 1998), and each one provides additional elements and modifications to the original CBR model. Every CBR system has a fundamental component, the

Casebase, which contains the cases previously encountered by the agent (or, in other words, the "problems" previously solved). A *case* in a CBR system can be represented in many ways, but most commonly it is a collection of feature-value pairs.

Features can be either symbolical or numerical. A symbolic feature has a finite domain: it can assume a limited amount of values, which are known and pre-determined. A numerical feature, instead, can usually assume any real value. Every CBR system uses a set of features common to every case in the Casebase, even if not all the feature values have to be specified for every case. Now that we defined features and cases, we then ask ourselves how we can compare two given cases. In other words: given two cases, we would like to assign a numerical value to their degree of *similarity*.

To accomplish this task, we can define a function $f(c_1, c_2)$ that has the following properties:

- ❖ $f(c_1, c_2) \geq 0$
- ❖ $f(c_1, c_2) = 0\ iff\ c_1 = c_2$
- ❖ $f(c_1, c_2) = f(c_2, c_1)$
- ❖ $f(c_1, c_3) \leq f(c_1, c_2) + f(c_2, c_3)$

Functions such as *f* take the name of *distance functions*. The smaller the distance between two cases, the greater their similarity, until the distance reaches zero if the two cases are equivalent (all the features have identical values). We will switch

between the terms distance and similarity, since we can compute one by computing the other.

Unfortunately, as we saw earlier, a case can contain features of very different nature, some not even numerical. Defining a function that computes the similarity (or distance) between two cases directly can be a daunting, sometimes impossible task. It is much easier to define local similarity functions that compute the similarity of each feature, and then to combine these local distances using a so-called *aggregation function*, which is usually a weighted average of the local similarities. Usually aggregation functions are normalized so that they return a value between 0 and 1, where a value of 1 represents two identical cases and 0 represents two ideally "opposite" cases. Choosing the weights for combining the local similarities can be considered a problem on its own; these weights can either be engineered and kept constant or can be learned and change throughout the system's execution. The aggregation function used by EXP_TEAM (Chapter 5) uses fixed engineered weights.

Now that we also defined similarity metrics, for any new case that gets presented to the CBR system, we can run a comparison against every other case in the Casebase and measure the similarity. We can then rank every case in the Casebase using these values, and therefore find the most similar case(s) to the one we are testing. The reason for doing so is that along with each case we are going to save the solution that we used for that particular problem. Now, for new cases, we can use the solution to similar problems we already solved in the past. This is a great

improvement, because now if an intelligent agent gets tested in a completely new environment in respect to the one it trained on, it can still perform well if it is able to find similarities between the test environment and the training one.

The CBR learning model follows the so-called "4 R cycle": Retrieve-Reuse-Revise-Retain. First, the system gets presented with a new problem for which we want to find a similar case already solved in the past; we then proceed *retrieving* one from the Casebase, using the distance function defined above. Once we found a case that fits our needs, we then go to the *reuse* phase where, as the name suggests, we reutilize the solution for the old problem to the new one. Not every system implements the so-called *revise* phase, but systems that do will modify the old solution to the new problem so that it better fits the small differences that can exist between the two. Finally, if the new problem can help solving future problems (if it is not exactly the same as a problem already in the Casebase), then it is added to the Casebase in the *retain* phase. These four processes exist in many different variants and implement different heuristics; we will show in detail in Chapter 4 the EXP_TEAM's CBR implementation.

2.3 Discussion

We saw how RL and CBR can be used in an intelligent system to implement learning mechanics. The two systems, while very different, can be used together to achieve greater performance. While RL learns which action is best to take at a given specific state of the environment, CBR can learn how to compare different states of the environment. This way, we can improve the learning rate of RL the following

way: whenever the agent has to pick which action to take, it can compare the current

of the state of the environment against its Casebase. If it can find a similar state seen

in the past, then it is likely that the available actions in the current state will provide

similar rewards to the ones we observed in the past for the similar state. This synergy

of RL and CBR is fundamental for the EXP_TEAM agent, and it is particularly

effective in EXP_TEAM+, as we will see in Chapter 4.

Chapter 3   Related Work

Whenever an agent commits a mistake, there can be many reasons behind it: the types of possible reasoning failure are numerous, and they are thoroughly discussed in (Cox, Introspective Multistrategy Learning: Constructing a Learning Strategy under Reasoning Failure, 1996). A reasoning failure of particular interest to us is *contradiction*. An intelligent agent, when performing an action, makes assumptions on what the resulting state of the environment will be after the action is resolved, we call this the *expected state* of the environment. Unfortunately, these assumptions are not always verified: in this case we have a contradiction or, equivalently, an *unexpected failure*. This kind of failures represents a great opportunity for the agent to increase its knowledge of the environment. To do so, the agent must be able to *explain* why the expected state and the actual state of the environment differed.

A very insightful distinction between Learning Goals and Achievement Goals can be found in (Cox, Perpetual self-aware cognitive agents, 2007). Once an unexpected failure event occurs, the agent must be able to answer a fundamental question: what is the reason of the failure that just occurred? Was it because of the unpredictability of the environment that the expected state didn't match the actual state, or could it be that the agent's knowledge is incomplete, or even flawed? While the answer to this question might seem irrelevant, it is indeed of incredible importance. This question gives way to deep, introspective anthropological discussions, and while diving into the philosophical aspects of AI is beyond the

scope of this thesis, it is again very interesting at least trying to see how we, as human beings, usually react to unexpected failures. After observing that the state of the world is not what we expected, most people (I admit falling into this category from time to time, as I think is the case with many other fellow engineers) find it easier to blame external and improbable causes as the reason to their erroneous expectations, rather than simply blaming themselves - *That marvelous algorithm you wrote does not behave as expected? It must be that old CPU of yours that suddenly decided to fault; I'm sure that if you try testing the algorithm on a different machine it will work perfectly* - While this line of thought can be correct once in a million times, it is nonetheless the wrong approach in the rest of the cases. Blaming ourselves for our unexpected failures is beneficial for two reasons: it saves us the time we would waste looking for a not-existing external culprit, and it also pushes us to increase our knowledge so that we can correct the mistake ourselves. An intelligent agent should behave in the same way, and once it encounters an unexpected failure it should blame itself and its own reasoning for it. Then, by generating a Learning Goal, it can improve its cognitive process, increase its knowledge about the environment, and hopefully avoid incurring in the same mistake in the future.

3.1 DiscoverHistory

Explaining discrepancies between the expected state and the actual state of the environment is a very active topic in AI research, and there exist already intelligent agents capable of such task, like DiscoverHistory (Molineaux, Aha, & Kuter, 2011; 2012). This intelligent agent relies on a Hierarchical Task Network

(HTN) planner, and for any given discrepancy, it generates a kind of explanation that we refer to as *deterministic*. The reason for labeling in this way the kind of explanations generated by DiscoverHistory relies on the fact that when presented twice with the same discrepancy, and assuming two equivalent states of knowledge, the agent will generate the same explanation. The agent's behavior is deterministic: it can be forecasted if we have perfect knowledge of the input.

DiscoverHistory generates a plan making assumptions about the conditions of the environment, since it cannot account for every plausible event that can happen. As always, things can and will go wrong at some point, and the plan will fail, presenting inconsistencies. This is when the DiscoverHistory algorithm gets triggered with the task of explaining the failure and resolve it. To do this, the algorithm makes modifications to the original plan, inserting events that would solve the inconsistencies. Often, the new elements inserted in the plan generate new inconsistencies that are then recursively resolved. The algorithm halts either when it found a satisfactory explanation, or when a maximum number of modifications has been made and the algorithm gives up. Very often, multiple explanations can exist for a given discrepancy, and in this case DiscoverHistory uses an Occam's razor heuristic and picks the explanation that requires the least amount of modifications to the original plan as the correct one. This heuristic introduces a bias in the DiscoverHistory algorithm that causes its predictability and which, even if very effective in many occasions, can also prevent reaching the correct explanation in many instances.

3.2 ARTUE

Recently, (Klenk, Molineaux, & Aha, 2012) present ARTUE, an intelligent agent that uses an updated version of DiscoverHistory to explain discrepancies in a real-time strategy (RTS) game environment. ARTUE is based on a modified version of the vastly used SHOP2 planner (Nau, et al., 2003), and uses an enhanced version of the DiscoverHistory algorithm discussed before.

As we will see in the next Chapter, RTS games represent a very sophisticated environment, where many unpredicted happen frequently. In the results of this work, we can clearly see how an intelligent agent with the capability of explaining such events has a clear advantage in respect to an agent that does not.

Both ARTUE and DiscoverHistory blame the unpredictability of the environment for any discrepancy that occurs. A different approach (this is the path we decided to undertake for this thesis' work) is for the agent to blame itself whenever its expectations do not match the actual state of nature. When the environment is highly unpredictable the agent should accept this as a fact, and should prepare to the fact that its knowledge is very limited and never complete. Whenever a discrepancy happens, the agent must consider it as an occasion to learn and to improve its own cognitive process, with the clear goal of being better prepared in the future. While it is easy to give an explanation to certain facts because we know that the likelihood of our explanation to be correct is very high, in other occasions it is not so easy. With the increasing of the free parameters in the

environment, the number of plausible events also increases dramatically, and so do the possible explanations to each one of these events.

This argument does not reject the validity of the cause-effect principle: we still believe that to each event there is one correct explanation. Although, this does not imply that the number of *plausible* explanation might be very high; the problem now is discerning among the plausible explanations which one is the correct one. Since the amount of information we have available is almost always limited, since the majority of complex environment are partially observable, then we have no better way to discern the correct explanation than taking an educated guess. A greedy approach, as in DiscoverHistory and later in ARTUE, would use some kind of heuristic to determine which explanation is most likely, and then will pick that one in a deterministic way. While this heuristic is often very effective, it can also perform badly in the case we have many possible explanations to a given discrepancy, and each one of them has similar likelihood. In Chapter 4 we will discuss how our approach is consistently different, and how we an intelligent agent can constantly learn about the environment it operates in, correcting over time its estimation of the likelihood of each explanation.

Chapter 4   Stochastic Explanation

In Section 2.1 we explained how an intelligent agent using the RL model can learn which action gives the best reward in a given state. This is done by testing all possible actions in a given state, and requires many training iterations before the system can operate efficiently. The typical RL paradigm is synthetized in Figure 1: RL Model.
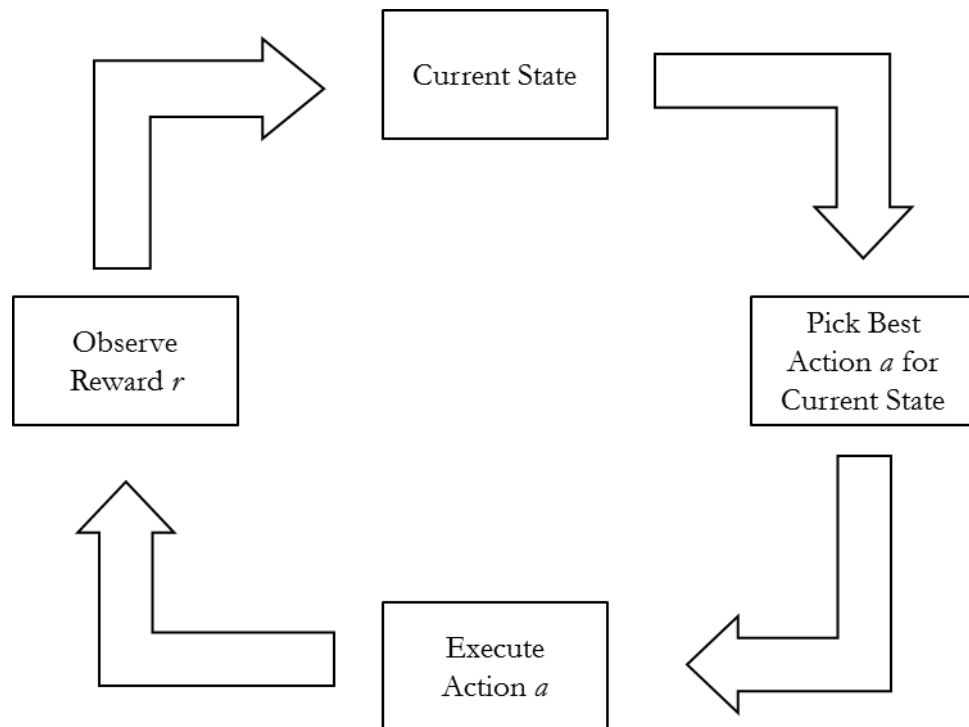

Figure 1: RL Model

This model works very well when the environment is deterministic: the reward r collected when performing action a in state s is constant over time. In other words, performing the same action twice in the same state will result in the same reward. While this assumption holds in many real-world applications, it does not in many

others. This is not the only limitation of the RL model: if the environment is vast, learning the effects of every possible action for each plausible state can be an extremely expensive, most likely unaffordable, task. In Section 2.3 we anticipated how we could integrate CBR to overcome this problem, now we will expand this idea to show how the synergy of RL and CBR can be used to create a more flexible and reactive intelligent agent.

As previously discussed, a necessary condition for RL to operate efficiently is to have knowledge of the current state of the environment. More accurately, an agent using RL needs to know the state of the environment in respect to its own interpretation of it. Unfortunately, broad environment can present so many plausible states that knowing them all can be impossible. However, if we were capable of finding similarities between two states, then we might infer that two states are substantially the same (even if not exactly the same), and therefore the rewards provided by executing a certain action in a given state should be similar, if not equal, if executed in a similar state. CBR is an ideal candidate to accomplish this task. We will have a Casebase containing the previous states, each one associated with the rewards provided by each action.

In Figure 2 we can see a representation of the extended RL model, comprehending a CBR module for retrieving past similar states from the Casebase. In case a sufficiently similar case cannot be found, then the current state is considered unprecedented, and it is added to the Casebase. In this case, RL would

then go ahead and pick an action at random for the current state, observing the reward and updating the case accordingly.
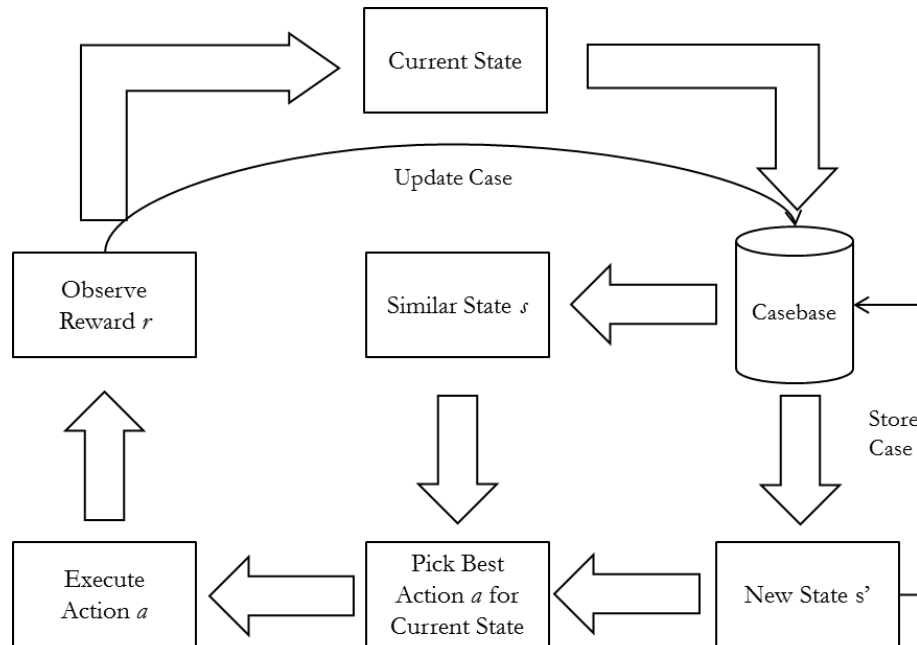
Figure 2: RL Model Extended with CBR

In the next sections we will show how we can use this extended model to explain anomalies in a stochastic domain, and how these explanations handle the unpredictability of such domains.

4.1 Learning From Mistakes

The necessary condition for being able to learn by mistakes is having the ability to recognize when a mistake was made in the first place. Whenever an intelligent agent takes an action, it makes a hypothesis of what the resulting state of the environment will be as a result of the action. We call this the expected state of the environment. If, instead, the resulting state is different than the expected state,

we have what we call a *discrepancy*: the expected state and the actual state differ. The process of identifying a discrepancy is not particularly challenging since it merely involves comparing actual and expected states. However, we would like to generalize the concept of *expected state* to be applicable to not only the resulting state after the agent executes an action, but more generally to the resulting state after the agent executed a sequence of actions required to reach a certain objective, which takes the name of *Goal State* or simply *Goal*.

This generalization lets us discuss explanations at a much higher level than before. The events we must explain, discrepancies, now do not involve a simple action, but they are intended as a broader concept. Sometimes, being too specific can prevent us to see the bigger picture; also, there are many instances in which the repercussions of our own actions on the actual state are visible only long after the actual mistake was made: blaming the last action the agent executed as the culprit for a discrepancy would be completely inadequate for us to understand what the correct explanation is. Also, among the possible anomalies we must consider the possibility that the goal we were trying to pursue was unrealistic.

This latter aspect is what interests us the most. An intelligent agent usually starts its execution knowing very little, if anything, about the environment it operates in; the objectives pursued by an agent having a very limited knowledge of the environment are often going to be poorly chosen, and presenting unrealistic expectations. When the agent's expectations are not met, we must then be able to explain this event by blaming the agent's cognitive process and limited knowledge,

which caused it to pursue a bad goal. This is what we do in EXP_GEN; then, in GENERATE_GOAL+ we show how the generated explanations can be used by the agent to avoid pursuing unrealistic goals.

4.2 Definitions

So far we discussed about explanations generically and safely assuming that a discrepancy has one, and only one, explanation. The reality is often different: a discrepancy usually has multiple possible explanations, and the correct one depends on the instance we are considering. In a stochastic domain the correct explanation to a discrepancy can also change over time. This is because the information we have is almost always limited, since we are considering an agent which operates in a partially-observable domain. For each discrepancy, we will then have a *list* of possible explanations, each one of them associated with a probability value that tells us the likelihood of that particular explanation to be correct for the given discrepancy. This probability cannot be known *a priori* but must be learned through experience, and this is where RL can be used. The first time we see a discrepancy, every explanation will have equal probability because the system never actually tested any of them in such occasion. Each explanation is associated with a *reaction*: a sub-goal that the agent tries to pursue to test the validity of the explanation. If the reaction succeeds, then the agent will consider the explanation as correct, raising its probability value; otherwise, it will decrease it.

In our representation, an explanation is merely a label: a description, more or less accurate, of the discrepancy it is trying to explain. It is not important if the

explanation has a meaningful name; in fact, we could label every explanation with a numeric ID. The fact that we give explanations a meaningful name is for us humans to better understand what the system is doing. If the system tells us that the correct explanation for a given discrepancy is *0*, or *A*, or *0xD*, we don't really know how to interpret this information. If instead the system tells us that the explanation is *Insufficient Resources* then we have a better chance to understand what is really happening. To the system, though, the two options are the same. What is important is that given a discrepancy, the system is able to find a set of explanations with their associated probability values.

We have now presented all the elements we need to define a Stochastic Explanation. Let E be the set of all possible explanations. For any set A, the notation 2^A is the power set of A: the collection of all subsets of A. A stochastic explanation is an element $\epsilon \in 2^{E \times [0,1]}$ such that ϵ is a probability distribution. That is, $\epsilon = \{(e_1, p_1), \dots, (e_n, p_n)\}$ such that each $e_k \in E$ and $\sum_k p_k = 1$ hold. A stochastic explanation therefore associates each explanation with a probability value; every case in the Casebase of the CBR module will contain a pair (s_i, ϵ_i) where s_i represents a previous state of the environment when a discrepancy occurred, and its respective stochastic explanation.

We then need to decide which explanation in ϵ we want to use for the current discrepancy. Our system will make this decision according to the probability distribution. After several iterations, RL will have learned the probability distribution of each stochastic explanation. An alternative approach could be consistently picking

the explanation that has the highest probability value; this approach has considerable drawbacks that will be discussed in the next sections of this Chapter. For now, we merely define this approach as *greedy*, and we denote the explanation having the highest probability value as $greedy(\epsilon)$:

$$greedy(\epsilon) = \max \arg_p \{(e_1, p_1), \dots, (e_n, p_n)\}$$

So far we assumed that the set of possible explanation is predetermined and immutable. In fact, learning such set is an entirely different problem, and a very challenging one. We leave this fascinating challenge for future work, presenting it more in detail in Chapter 7.

4.3 The Wargus Domain

Every topic discussed so far was domain-independent, thus applicable to most situations. Now we will present the specific domain used for our application, and most of the discussion from here on will refer to this particular domain.

The domain of choice for our experiments is Wargus, an open source game clone of the popular Warcraft II by Blizzard Entertainment™ and based on the Stratagus open source engine. Wargus is a Real-Time Strategy game where two or more opponents battle for victory; we restrict our study to the case of two opponents playing the game. As many other RTS games, Wargus presents two aspects: combat and resource management (Jaidee & Muñoz-Avila, 2012). While the latter aspect is very intriguing and represents a challenge on its own, we concentrated on the former aspect of the game for our experiments. Tasks like building structures,

gathering resources, producing new units, etc. are not covered in our experiments: the two opponents begin every scenario with a predetermined number of units that they can use for battle; whenever one of the two opponents does not have any more units left, the scenario is over and the other player is the winner. In Figure 3 we can see a typical Wargus Combat Scenario, where different units are preparing to engage in combat. The unit armors are colored differently (red and blue in the figure) so that players can distinguish between their units and the opponent's.

Figure 3: Sample Wargus Combat Scenario

The game units follow a rock-paper-scissors mechanism that keeps them balanced: every unit type has other unit types it is strong against, weak against, and fairly even against. Also, the units are divided in two races, orcs and humans, but every unit in a race has its equivalent in the other race, making the two races substantially the same. Nonetheless, there are some minor differences, and to

eliminate any possible bias, in our experiments both players use human units only. The units are divided in three main groups: flying units, land units, and sea units. This introduces an additional strategic element to the game since some units have restrictions on the other kinds of unit they can attack (e.g. a knight cannot attack flying units). Some units can also be upgraded to their more powerful version; for example, an *archer* unit can be upgraded to become a *ranger*. This will improve some of the unit's stats like health points, attack power, etc.

4.4 CBR at work

Whenever a discrepancy is detected, the system must provide a stochastic explanation for it. A plausible approach could be storing a set of 1:1 correspondences between previous states of the environment when a discrepancy happened and their respective stochastic explanation. However, this approach would be extremely expensive and most likely inefficient, since it will be almost impossible to find two states that are exactly the same. A much more feasible approach is using CBR techniques defining local similarity metrics to compare several features from the environment, and then combine them in a global similarity metric using a weighted aggregation function.

As soon as the expected and the actual state of the environment differ, the agent takes a *snapshot* of the actual state, and extracts features from it. These features are domain dependent attribute-value pairs. In Wargus, we currently use five features: the Euclidean distance between attacker and target, the difference of health points between the attacker in the retained case and the query case, the same

difference for the target, and finally the types of both attacker and target. When presented with a query case, the system checks the similarity of it with all the previous cases stored in the Casebase, computing the local similarity $sim_i(x_i, y_i)$ between pairs of features x_i, y_i (Ricci & Avesani, 1995). The global similarity metric between two discrepancies is then computed with an aggregated similarity function SIM_{agg} defined as:

$$SIM_{agg}(X, Y) = \sum_{i=1}^{n} \alpha_i \, sim_i(x_i, y_i)$$

This similarity and its weights α_i are also domain dependent. If the aggregated similarity value is above a user-defined threshold, we consider the discrepancy as "already seen" in the past, thus we can use its stochastic explanation in the current context. Otherwise, the discrepancy is considered as "unprecedented" and the case:

$(state\ of\ the\ environment, equiprobable\ stochastic\ explanation)$

is retained in the Casebase for future use. In this latter case, the probability values will be initialized all equal, and we refer to this situation as an equiprobable stochastic explanation. The system does not know anything about the new discrepancy, therefore the likelihood of each explanation to be correct is the same; once the explanations will be tested (by observing the outcome of the reaction) the probability distribution of the stochastic explanation will be updated accordingly. If, instead, the system was able to find a sufficiently similar discrepancy stored in the Casebase, then it will use the same stochastic explanation for the current discrepancy, pick an explanation and observe the reaction as usual.

4.5 Motivating Example

In Section 4.3 we presented the domain our intelligent agent will be operating in. All the concepts previously presented (e.g. the rock-paper-scissors mechanism) are initially unknown to the agent; that is, the agent has no knowledge of what unit it should use to attack a given enemy unit, or if using an upgraded unit could be preferable. Moreover, the agent cannot see autonomously if an enemy has an *environmental advantage.* We use this term to identify an enemy unit that is using the environment to gain some benefit; in our experiments, we represent this condition with an enemy unit surrounded by trees. In this situation, a short-ranged land unit like a *knight* cannot attack units that in normal conditions it would be able to, like an *archer.*

We will now show a motivating example, demonstrating how an agent can benefit from the ability of explaining anomalies. In our experiments, the agent only generates one kind of goal: *Kill Unit A using Unit B*, and the only knowledge the agent has at the beginning of its execution is how to reach this goal, by executing the simple policy *Attack Unit A using Unit B*. Obviously, we include under this policy the rather simple knowledge of the action *Move unit B to position (x,y).* The agent always keeps an optimistic outlook on the goals it generates: it always expects the attack action to be successful and to see as a result of this attack that *Unit A* is now dead. If this condition is not met, we have a discrepancy which triggers the *Explanation Generator* module that must then find an explanation to why the attack failed.

In this experiment, as well as in Chapter 6, we will consider a fixed set of three possible explanations E: *unit B was not upgraded* (e_1); *unit B is a bad choice to attack unit A* (e_2); *unit A has an environmental advantage* (e_3). Every explanation has a *reaction* associated to it, which is also predetermined and fixed. Whenever an explanation gets picked by the Explanation Generator, the system executes the corresponding reaction monitoring the outcome. If the outcome of the reaction is successful, then the explanation is considered to be correct and its probability is raised, otherwise we do the opposite. The reactions to the three explanations are, respectively: *attack unit A with upgraded version of unit B* (r_1); *attack unit A with a better unit* (r_2); *attack unit A with a ranged or a flying unit* (r_3). Regarding reaction r_2, the system does not know from the start which unit is *better* to use to attack unit A, but it will keep track of the battle outcomes for every time it performs an attack, learning with time the rock-paper-scissors mechanism explained previously.
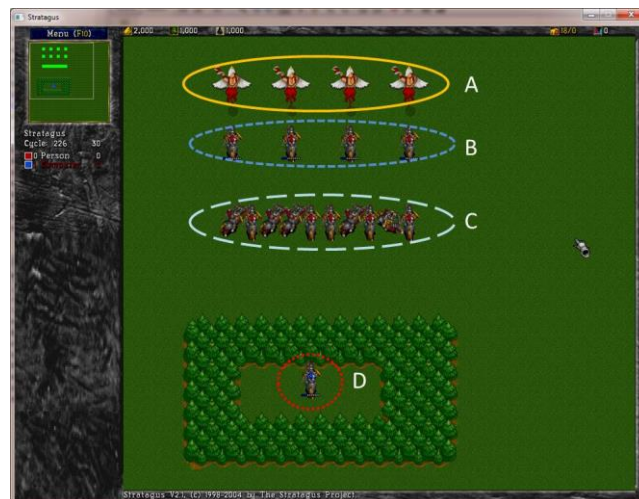

Figure 4: Scenario 1. A are the griffins, B are the paladins and C are the knights. D is the enemy paladin surrounded by trees and hence inaccessible for land units.
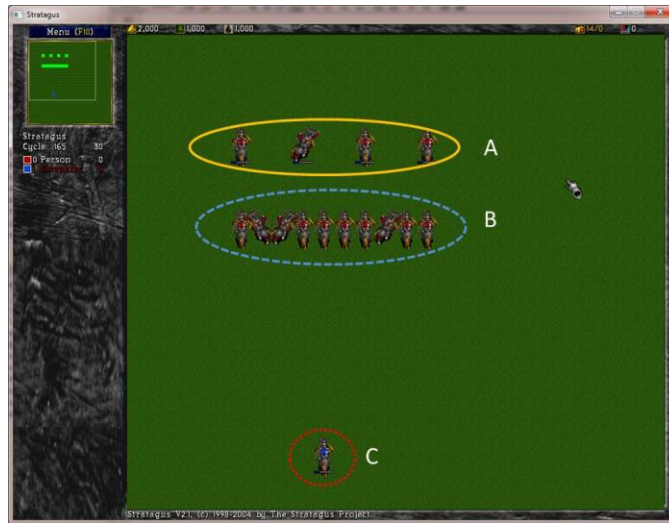
Figure 5: Scenario 2. A are the knights, B are the paladins. C is the enemy paladin, now accessible from land units.

The motivating example is composed of two scenarios, which are both played for 50 iterations. The knowledge acquired during the execution of the first scenario is kept and reused during the execution of the second scenario. The agent has no perception of the change of scenario: we treat this as an unexpected event, very similar to the kinds of unpredictable events that can always happen in a stochastic domain. In the first scenario (See Figure 4), the agent has several knight units, some paladin units (which is the upgraded version of a knight), and also several griffin units, which are powerful flying units that can attack land units. The enemy AI, which is a passive script that only attacks back when attacked, only controls one paladin, surrounded by trees. We say that this paladin has an environmental advantage. During the execution of the first scenario, whenever the agent attacks the

32

paladin with a land unit, it fails. With time, it learns that the right explanation for this fact is that the enemy has an environmental advantage, and therefore the right thing to do is to attack with a griffin. After several iterations the system will learn a stochastic explanation such as $\epsilon_1 = \{(e_1, p_1), (e_2, p_2), (e_3, p_3)$ where $p_3 > p_1$ and $p_3 > p_2$ hence $greedy(\epsilon_1) = e_3$ holds.

In the second scenario (see Figure 5), the conditions are almost unmuted, except that now our agent does not have any griffin units, and there are no more trees surrounding the enemy paladin; this can happen in a regular RTS game, because the *peasant* units can chop trees down. Now, when attacking the paladin with a knight, the agent fails but the correct explanation is now different: the knight is not upgraded (i.e. explanation e_1) therefore we should have attacked with a paladin. At first, when we switch to the second scenario, when we fail to kill the enemy paladin, the system will have a high probability associated with the explanation environmental advantage (i.e. explanation e_3). This means that the greedy explanation will always pick environmental advantage, as long as the probability is higher than the others. Therefore, a greedy approach would have to wait until the probability of environmental advantage falls below the probability of the other explanations, making a lot of mistakes until then. An agent using stochastic explanations, instead, will pick a different explanation sooner, thus learning that the environment's conditions changed much quicker. If the agent still had griffin units in the second scenario, then both explanations e_3 and e_1 would succeed, since both their reactions would provide a positive outcome. In this case, Stochastic Explanations would learn

a probability distribution that depicts these conditions. Greedy would keep using explanation e_3 instead, since it never stopped working. While the performance of the two approaches would be similar in this case, we need to consider that Stochastic Explanation's representation would be closer to the state of nature. If, eventually, the agent cannot use griffins anymore, then an agent using Stochastic Explanations will be able to recover rapidly, while an agent using Greedy Explanations will suffer greatly, taking a much longer time to re-establish good performance.

This is the reason that motivates the experiments we present in Chapter 6. If an intelligent agent operates in a stochastic domain, then greedy explanations will have a hard time reacting effectively and in a timely manner to the sudden changes that happen in such domain.

4.6 EXP_GEN

We will now present a pseudo-code for the EXP_GEN algorithm the system uses to retrieve and generate explanations for any given discrepancy. The algorithm requires five elements in input: the Casebase CB which contains the discrepancies previously encountered (CB will be empty at the first iteration of the system), the policies Π to execute the goals as defined by their reactions (i.e., pre-coded instructions of what to do next), the pre-determined set E of (explanations, reaction) pairs $<e,r>$, the similarity threshold σ, and the number of goals θ.

```
EXP_GEN(CB, Π, E, σ, θ)
1. GoalsInProgress ← ∅;  ExplanationsInProgress ← ∅,
   FailedGoals ← ∅
2. while true
```

```
3.     // Get the current state from the  environment
4.     s ←GetState()
5.     if ScenarioOver
6.        return CB

7.     for-each g ∈ FailedGoals
8.     //Generate Explanations for discrepancies
9.        c  ←  NewCase(g,s)  //Create query case
10.       c' ← GetNearestNeighbor(CB, c) //Retrieve NN(c)
11.       if (Similarity(c',c) ≥ σ)
12.       // Consider the query case as the same case of c'
13.          e ← PickStochasticExplanation(c'.Explanations)
14.       else
15.          // Consider case c as unprecedented
16.          CB.retain(c)
17.          for-each <e,r> in E
18.             //initialize as equiprobable
19.             c.addExplanation() ←  (<e,r>, 1/|E|)
20.          e ← PickStochasticExplanation (c.Explanations)
21.       ReactToExplanation(e, Πₑ)
22.       ExplanationsInProgress.add(e)

23.       for-each e ∈ ExplanationsInProgress
24.          UpdateExplanationState(e)

25.          if (e.state == success)
26.             RaiseProbability(e.p)
27.          if (e.state == failed)
28.             LowerProbability(e.p)
29.          if (e.state != in_progress)
30.             ExplanationsInProgress  =
                          ExplanationsInProgress – {e}
31.       //end for-each
32.       if GoalsInProgress < θ & ExplanationsInProgress =∅
33.          g ← GenerateGoal()
34.          PerformGoal(g, Π_g)
35.          GoalsInProgress.add(g)

36.       for-each g ∈ GoalsInProgress
37.          UpdateGoalState(g)
38.          if (g.state == failed)
39.             FailedGoals.add(g)
40.          if (g.state != in_progress)
41.             GoalsInProgress = GoalsInProgress – {g}
42.    //end while
```

After resetting Goals and Explanations in progress at Line 1, the algorithm enters a loop, which will be terminated when the scenario ends (Line 6). At Line 4, we update the agent's state of the environment representation; if the scenario is over, the algorithm returns the updated Casebase.

At Line 7, the algorithm loops through the failed goals (if any), generating a new case for each one. Then, at Line 10, the algorithm looks for the most similar case to c in the Casebase. If the similarity of c' and c is greater than σ (Line 11), the system picks the explanation from c' at Line 13, otherwise the case c is considered unprecedented. It is then added to the Casebase (Line 16), its explanations are initialized with the given input set E all having the same probability $1/|E|$, and an explanation is then picked at Line 20. The function *PickStochasticExplanation* takes a stochastic explanations *c'.Explanations* as input. It picks an explanation *<e,p>* according to the probability distribution in *c'.Explanations*.

At Line 21 we call the function *ReactToExplanation*, which takes in input the explanation e and the reaction associated to it in order to perform the reaction. Then, at Line 22 we add this explanation to the ExplanationInProgress collection.

Lines 23-30 handle the update of the state of each explanation in progress. In case the reaction succeeds, which means the explanation was correct, the probability of the explanation e is raised. If the executing the reaction fails to achieve the goal, the probability of *e* is reduced. Finally, unless the reaction associated with explanation *e* is still in progress, we remove it from the *ExplanationInProgress* collection at Line 33.

The probability value of an explanation is merely the ratio of successes over the total number of trials, with an additive term called *booster* value.

$$P(explanation) = \frac{Times\ Succeeded}{Times\ Tested + 1} + multiplier \cdot boosterValue$$

There are two kinds of booster values: a *reward booster* value and a *punishment* one. When an explanation is correct, its probability is increased by the reward booster value, and similarly decreased by the punishment booster value when incorrect. The multiplier will increase by one for each consecutive success/failure and will be reset to 0 (no booster) whenever a success-failure or failure-success pattern occurs. If the explanation is tested another time, and it results correct once again, the multiplier will be now 2 and therefore the probability will be raised more. The two booster values are user-defined parameters and they can influence the reaction time of the algorithm. In our experiments, we use a booster value of .12 to reward correct explanations and a booster value of -.01 to punish incorrect ones.

We generate a new goal only if there are no explanations in progress (Line 32) and only if there are no more than θ goals already in progress; in our experiments, we set $\theta=1$. This makes the system pursue one goal at a time, eliminating possible noise from the performance analysis we will present in Chapter 6. The GenerateGoal function generates only one kind of goal: kill unit A with unit B, where A and B are randomly chosen (so there are many goals that can be pursued, one for every pair (A,B). This goal then gets started by the *PerformGoal* function, which takes the goal and its associated reaction as input (Line 34). The goal is then

added to the *GoalsInProgress* collection (Line 35). At Line 37 we handle the update of the state of the current goal. For each goal g, if g failed then we add it to the GoalsFailed collection. And finally, unless the goal is still in progress, we remove it from the *GoalsInProgress* collection (Line 41).

4.7 GENERATE_GOAL+

The knowledge we collect in this way has the ultimate goal of preventing us from committing the same mistakes in the future. To do this we need to improve the goal generation code, making it search the Casebase for past mistakes that involved goals similar to the one we are trying to pursue, and this is what the Generate_Goal+ function does. If we find that we indeed made mistakes in the past while trying to pursue a similar goal, then it must mean that the goal should not be pursued and that we need to modify it. Notice that this process will not prevents the agent to commit mistakes in the future, because the environment is stochastic and unexpected events can and will happen, but nonetheless this procedure will dramatically decrease the number of times the agent will try to pursue unrealistic goals. The following pseudo-code describes the Generate_Goal+ function.

```
GENERATE_GOAL+(CB, τ)
1. g ← CreateGoal()
2. if FindDiscrepancies(g, CB) > τ
3.     PickBestGoal(g)
4. return g
```

At Line 1, the function *CreateGoal* will generate a domain-specific goal. As explained before, the goal generated will always be *KillUnit(A,B)* where A and B can

be any unit. At Line 2 the function *FindDiscrepancies* will scan the case base CB, looking for previous failures that involved the same kinds of unit that were picked for g. This is a retrieval operation from the case base that involves a similar, but more relaxed, similarity metric. Here we consider only the unit types as features to compute the similarity.

If *FindDiscrepancies* can find more than τ previous failures, where τ is a user-defined threshold, then the algorithm calls *PickBestGoal* at Line 3. This function will look for the best unit to attack the target among the agent's units. To do this, we rely on knowledge acquired during the execution of the system. Recall that the system starts its execution knowing nothing about the rock-paper-scissors mechanism of the game, but improves its knowledge during its execution. Therefore, this algorithm will improve its performance over time, effectively learning from past mistakes.

4.8 CombatPlan

So far we considered the case where an intelligent agent pursues one goal at a time, observing the outcome and explaining any discrepancy that might occur. This behavior is perfect for training conditions, because it eliminates the noise that occurs if the agent pursues multiple goals at a time. In fact, considering the Wargus domain, if we attack a *knight* with a *footman*, the normal outcome is that the knight wins the battle. If we attack a knight with ten footmen, though, the outcome will be positive for the footmen. Handling these conditions would require a huge increase in the complexity of the agent.

If, instead, we train the agent on the assumption of one unit attacking one unit, with no external units involved in the combat, we eliminate the noise and we will be able to use the acquired knowledge in more complex scenarios after the training phase. The CombatPlan algorithm uses the knowledge acquired during the training phase of the agent to create a *combat plan*, which consists in assigning a target for each of the agent's units. Using the explanations created during the training phase, the CombatPlan algorithm is able to pick, for each of the agent's units, which is the best target among the enemies units. Once the combat plan is generated, all the units attack their target simultaneously. This simulates more realistic combat situations, where a player has to face multiple enemies at the same time. Once every unit attacked their target – that is, either the unit or the target is dead – a new combat plan is generated for the remaining units, and is then executed; this process continues until the scenario is over.

```
CombatPlan(myUnits, enemyUnits, CB)
1. while ! scenarioOver
2.    plan = newPlan()
3.    for-each unit ∈ myUnits
4.        target ← pickBestTarget(unit, enemyUnits, CB)
5.        plan.add(unit, target)
6.    executePlan(plan)
```

The CombatPlan function takes as input the agent's list of units, the enemy's list of units, and the Casebase containing the knowledge acquired during the training phase. At Line 1 the function goes in a loop that will be executed until the scenario is over and either the agent or the opponent wins the battle. At Line 2 we initialize the combat plan, creating a new, empty plan. At Line 3 we loop through the agent's

units, and for each of them we call the function *pickBestTarget*. This function takes the current unit, the list of enemy units and the Casebase in input, and uses the knowledge in CB to compute what is the best target among the enemy units list for the current unit. The way this function accomplishes its task is based on the Generate_Goal+ algorithm explained previously. Once the function returns, the association (unit, target) is added to the current combat plan (Line 5). After each one of the agent's units has been assigned to a target, at Line 6 we execute the plan: all the agent's units will simultaneously attack their respective target. If, after the attack, there are still units in the scenario, the algorithm executes the loop at Line 1 again, as many times as necessary.

It is important to point out how the training phase is crucial for the performance of this algorithm. In fact, if the knowledge contained in the Casebase is insufficient or incorrect, the function pickBestTarget at Line 3 will not be able to pick a good target for the agent's units, and this will result in a bad plan. In fact, if the algorithm does not have enough information to reach an optimal decision, then it will randomly assign the current unit to a target. Similarly if the knowledge is incorrect due to noise during the training phase, the pickBestTarget function will then pick the wrong target with disastrous effects on the efficacy of the combat plan. This tells us how the training phase is critical for producing a good plan; in Chapter 6 we will show how the algorithm performance depends on the amount of training data collected.

Chapter 5   Implementation

In this Chapter we want to dig deeper in the technical aspect of Stochastic Explanations. We will present how the EXP_GEN agent works, what are its parts and how they interact with each other.

5.1 Agent-Stratagus Interaction

Perhaps one of the most interesting aspects for somebody who has to implement a Java Wargus bot is how to communicate with Stratagus. The *Proxybot* class, a Java reimplementation of GameProxy.cpp by the Georgia Institute of Technology, handles the communication via socket. It exposes a rich, high-level API to access the current state of the game and to modify it. For example, the function *attack(attackerID, targetID)* can be used to attack the enemy unit with the specified targetID with the agent's unit having id attackerID. At the lower level, Proxybot will send a message to Stratagus via socket, specifying what it must do. Stratagus will then send an acknowledgement. The bot programmer does not need to worry about the low-level communication, and can merely use the higher-level functions exposed by the Proxybot class to interact with the environment.

5.2 Domain state

It is up to the developer to maintain a consistent view of the environment, and keep it updated to the actual state. To do this, the WargusStateImporter class exposes the function *getGameState*. However, our implementation uses an additional framework (Jaidee & Muñoz-Avila, 2012), which provides an extra layer of

abstraction and makes it easier to access the state of the environment. This framework provides a set of wrapper classes that make the interaction with Wargus much easier, providing also routines for handling the main game loop. In the rest of this Chapter, we will concentrate more on the concepts related to bot development and the EXP_GEN algorithm in particular, rather than digging into the details of the mechanisms that handle the game-flow.

The additional framework we used for the EXP_GEN bot exposes the *updateGameState* function, which throws a particular *GameOverException* when the scenario is concluded. To keep the game state updated, we simply need to call this function in a loop, always checking if the GameOverException gets raised to verify the Scenario did not end in the meantime. In the next Section, we will see how the EXP_GEN bot handles this task.

5.3 The Agent's Loop

Using the framework presented earlier to write a Wargus agent, we must start by extending the WargusTeam class. The constructor of this class requires an instance of the Proxybot class as parameter; also, this class is abstract, and has abstract methods we must override when extending it, in particular *assignCommand*. This method is called continuously during the execution of a scenario; in fact, it is called from the *runGame* method of the MainLoop class. We do not really need to care much about how and when this method gets called; what we need to be aware of is that this is the method where we need to implement the logic of our agent.
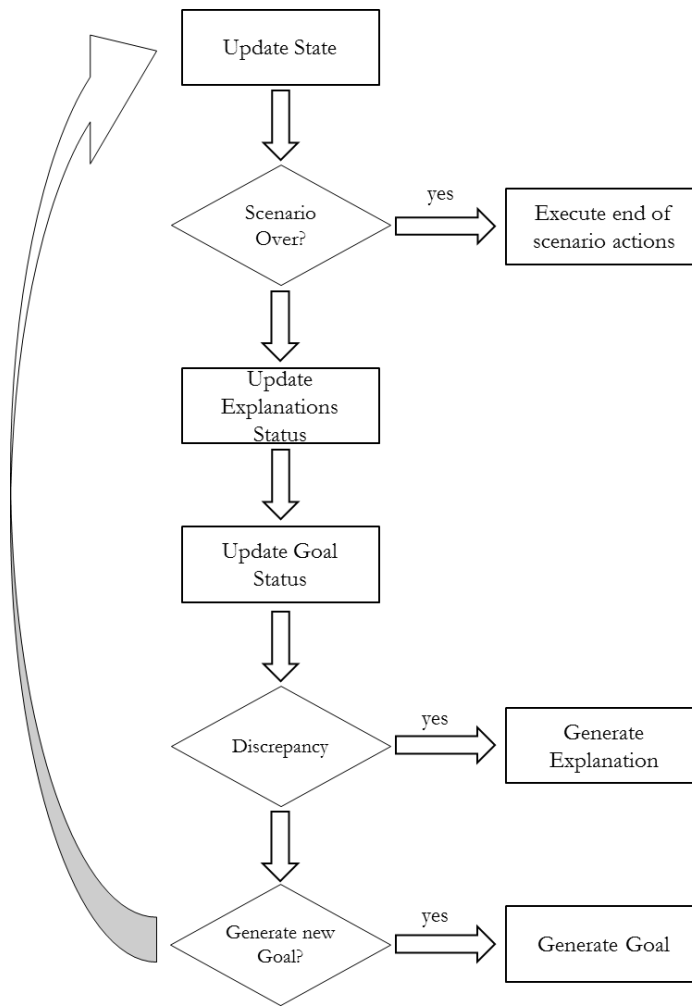
Figure 6: EXP_GEN bot loop

Figure 6 shows the logic of the EXP_GEN agent, which reflects the pseudo-code we presented in Section 4.6. The figure roughly represents the actions taken by the bot in the *assignCommand* function, excluding technical details and optimizations. The first action we must take is to update our representation of the state of the environment, and check if the scenario is now over. This can be done by checking if the updateGameState raises a GameOverExpection. If so, we will then execute any action we need to take at this specific moment: this might include gathering statistics,

instantiating new objects, resetting counters, and so on. If the scenario is not over, EXP_GEN then proceeds updating the status of any pending explanation that was generated in the past. This involves observing the status of the reaction associated with the explanation, and checking if it was successful; we then must update the stochastic explanation probability distribution accordingly.

EXP_GEN then checks the status of every pending goal; note that in Figure 6 we are considering the simple case of a bot that purses only one goal at a time. If the goal caused a discrepancy, we then invoke the Explanation Generator to find an explanation to this event. Note that we will check on the status of this explanation the next time the loop gets executed.

Finally, the agent will check if it is necessary to generate a new goal. The answer to this question depends on the number of concurrent goals our agent is set to pursue (in our case only one) and on the status of the previously generated goals. If the agent decides to generate a new goal, it invokes the Goal Generator to do so. Regardless if a new goal was generated or not in the last step, the loop will then restart from the beginning and we will update the game state once again.

5.4 Detecting Discrepancies

Before we can talk about the Explanation Generator, we first must discuss how to detect discrepancies. When executing the main loop the first time the agent will go straight to the goal generation step, since there are no explanations to check nor previous goals to update, therefore no discrepancies are possible. To generate a new goal, EXP_GEN picks a unit at random and invokes the Goal Generator, which

must then generate a goal for the random unit. The *GoalGenerator* class is rather simple: since EXP_GEN only generates goal of type *KillUnit* (Section 4.5), all that we need to do is to pick a random target among the enemy's units and assign it to our unit. The policy needed to achieve this goal (the sequence of actions the agent needs to take) is known and predetermined. Once the Goal Generator completed its task, the loop restarts from the beginning.

Once again, there are no explanations to update, since the agent did not incur in any discrepancies yet. However, now we do have to perform the Goal Update step. This step is executed using the *updateGoalsState* method from the GoalGenerator class. Let us assume that the goal failed because we made a poor choice selecting a target for our unit: the status of the goal would then be *FAILED*. Therefore, we detected a discrepancy. The goal is then added to the FailedGoals collection, which can be accessed invoking the *getFailedGoals* method in the GoalGenerator class.

5.5 The Explanation Generator

Once we detect a failed goal, we can use the *generateExplanation* method from the ExplanationGenerator class to generate a stochastic explanation for the failure. This method will instantiate a *StochasticExplanation* object using the failed goal, and will then try to generate an explanation.

Before we talk more in detail about the explanation phase, let us present the ExplanationGenerator class first. This class is a singleton: it cannot be instantiated traditionally since it has a private constructor. To get a reference to an

ExplanationGenerator object we must use the *getInstance* function exposed by the class. The reason for this is that there should be only one explanation generator instance for an agent. Also, making the class a singleton lets us access it statically, reducing the complexity of the code.

The ExplanationGenerator class maintains three data structures: the Casebase of discrepancies states and their stochastic explanations (also called *Explanation Table*), a list of failed explanations, and a list of explanations in progress. Every time we need to generate an explanation for a failed goal, we will run a nearest neighbor search against the cases in the Casebase to verify if a sufficiently similar failed goal has been observed in the past.

Going back to the previous example, when the first discrepancy is detected the Casebase is going to be empty therefore there won't be any neighbors to check. In this case, the stochastic explanation would be initialized with the given set of possible explanations (Section 4.6) all having the same probability value. This is called an equiprobable stochastic explanation. We will have to pick an explanation at random since we have no additional information. This new stochastic explanation is added to the Casebase and the explanation we picked from the set is added to the explanations-in-progress list. Next, depending on the agent's implementation, we can decide to generate a new goal right away, or we might prefer to wait and see the outcome of our explanation before we continue. In our experiments, we prefer to wait in order to reduce possible noise that can occur if newly generated goals interfere somehow with the outcome of the explanation.

When the agent's loop gets executed again, the explanation state needs to be updated. This is done intuitively via the *updateExplanationsState* method provided by the ExplanationGenerator class. If the state of the explanation is still IN_PROGRESS, nothing needs to be done and we give more time to the explanation to terminate its reaction. If the state, instead, is SUCCESS, we will raise the probability of the explanation we picked in the stochastic explanation's probability distribution; when the outcome is not positive and the explanation turns out to be incorrect, we will lower the probability. There are several states that indicate failure: FAILED, TIMEOUT, and IMPOSSIBLE. The FAILED state is reached when the reaction was executed correctly but the outcome is not satisfactory; the TIMEOUT state is reached after the outcome is not observable after a predetermined time threshold; the IMPOSSIBLE state is more subtle: It indicates that the reaction associated with the chosen explanation could not be executed for some reason (e.g. if the chosen explanation was Environmental Advantage and there are no flying/siege units available to attack). While the first two failure states are also used for goals, the IMPOSSIBLE state is peculiar to explanations. The IMPOSSIBLE state is useful for two reasons: it distinguishes normal failures and impossible conditions, and it improves the speed of failure detection. In fact, if we did not have the IMPOSSIBLE state, we would have to wait for the explanation to reach the TIMEOUT state before we can detect that it failed.

Supposing that the explanation we gave to our discrepancy terminated, we observe its outcome and we adjust the probability distribution accordingly. Next, the agent will generate a new goal, observe the outcome, explain any discrepancy that

might occur, and so on. This cycle continues until the scenario is over. When we detect a discrepancy and the Casebase is not empty, the generateExplanation function will run a nearest neighbor search. This search is executed by the *findNearestNeighbor* function inside the ExplanationGenerator class. This function will compare the states of the environment where the current discrepancy and the past discrepancy happened using an aggregated similarity metric, implemented in the *getSimilarity* function of the *StochasticExplanation* class. This function will output a number between 0 and 1, indicating the similarity of the two states; 0 indicates two completely different states, while 1 indicates two completely identical states. The function will return the state in the Casebase that has the highest similarity value with the current discrepancy. If the similarity value is above the predefined value SIMILARITY_THRESHOLD (0.7 in our experiments), then the two states are considered sufficiently similar and we can use the previous case's explanations for the current discrepancy; otherwise, we will consider the new discrepancy as unprecedented, adding it to the Casebase, and initializing its Stochastic explanation as equiprobable like before.

5.6 Other Elements

We discussed goal generation, discrepancy detection, and explanation generation: these are the main components that control the execution of EXP_GEN. Now we want to briefly introduce the other classes that are part of the agent, describing their functionalities.

A Goal has a State, a Policy and Actors. We discussed previously about what states and policies are. An Actor is an abstract entity that is involved in executing the goal's policy. For example, a KillUnit goal has two actors: an attacker and a target. These actors, in EXP_GEN, are always Wargus units; nonetheless, if we would change the domain, actors would represent different kinds of entities.

A Reaction is a particular Goal. It is associated to an explanation and it usually has a longer timeout value. The difference between reactions and goals is mainly semantic rather than practical. Also, while the status of goals is observed by the Goal Generator, the status of reactions is observed by the Explanation Generator; the way these two classes handle this task is similar, but with some differences like the "IMPOSSIBLE" state for reactions that we discussed previously.

The EXP_GEN agent has no knowledge of the rock-paper-scissors mechanism that balances the units in Wargus (Section 4.3); during its execution, the agent slowly learns this mechanism and stores the acquired knowledge in a *Combat Affinity Table*. This table, implemented in the CombatAffinityTable class, contains a list of Combat Affinities, one for each possible combination of attacker-target (e.g. footman-archer, footman-footman, knight-paladin, etc.). These affinities are initialized to 0, which is the best possible value: an affinity value of 0 means that the attacker is very strong against the target. When an attack goal gets generated, after specifying the attacker and the target, the agent increments the counter that keeps track of the amount of time that particular combination of units got picked. If the attack fails, and the selected explanation for this failure is *wrongUnit*, the combat

affinity of this combination will be updated, incrementing the failure count. The affinity level of a particular attacker-target combination is given by the following formula:

$$Affinity(A, T) = \frac{Failures(A, T)}{TimesTested(A, T) + 1}$$

The affinity is merely the ratio of failures over the number of tests of the particular combination (A, T). The +1 is an arbitrary additive term needed for scaling reasons. Its function is to prevent the affinity value from jumping immediately to 1 (worst affinity) if the first test results in a failure. The Combat Affinity Table is used by the GENERATE_GOAL+ (Section 4.7) and by the CombatPlan (Section 4.8) algorithms to make a more informed choice on which target to assign to each unit.

The class UnitType contains functions used by other classes to retrieve unit types, indices and IDs. These functions are all statically accessible, and their main purpose is to map index values with their respective unit types in the game.

The class Utils is a collection of static functions to obtain a reference to specific units in the scenario more easily.

Chapter 6　Empirical Study

In this Chapter we will compare the performance of an agent using stochastic explanations against two different agents: the first always picks the *greedy* explanation among the set of possible explanations; the second picks *randomly* among the same set. We claim that the agent using stochastic explanations will react faster to abrupt changes in the environment. Then, we will show how an agent that learns from past mistakes greatly outperforms an agent that does not perform this process. Finally, we present an experiment to show how the performance of an agent that learns from past mistakes increase significantly with the number of discrepancies it is able to observe.

6.1 Experimental Setup: EXP_GEN & GENERATE_GOAL+

In our experiments with the Wargus domain, we consider only one possible goal: *kill unit A with unit B*, with A and B chosen randomly. Whenever a goal is not accomplished (e.g., unit B is not killed), this generates a discrepancy; in our scenarios, we used a similarity threshold $\sigma=0.7$, and the explanation for each discrepancy is chosen from a predetermined set of three possible explanations: *Unit not upgraded*, *Wrong Unit*, and *Environmental advantage* (as seen in Chapter 4). The first represents the situation where the target should be attacked with an upgraded version of the unit that was used originally (e.g. ranger instead of archer). The second is probably the most intuitive explanation, and means that the attacker should have been a completely different unit (e.g. knight instead of peasant). Finally, the latter explanation, Environmental Advantage, represents the situation where the enemy

has some kind of obstacle protecting it from being attacked by closed-range land units and should therefore be attacked with flying or ranged units. In the description below we refer to *team A* to the one controlled by our AI and *team B* to the opponent.

As in other works on explanations (Klenk, Molineaux, & Aha, 2012), our scenarios are hand-crafted to reduce noise and present more accurately the effectiveness of our theories. We will present the results from three scenarios, each composed by two parts. In the first part, the system acquires knowledge about the environment, the discrepancies that can happen within it, and their corresponding explanations. In the second part the scenario is slightly modified, so that the previously acquired knowledge is now flawed and not effective for the new conditions. We will show how Stochastic Explanations react to these events in comparison with Greedy Explanations and Random Explanations.

In the first scenario, part-1 presents team A consisting of archers and rangers attacking team B which consists of rangers only. When archers are selected to accomplish the goal, they will fail because rangers are stronger. The correct explanation for this kind of discrepancy is Upgraded Unit. In the second part, we also have griffins available, and the enemy rangers are now surrounded by trees. When the rangers attack they will sometimes fail. The correct explanation becomes Environmental Advantage; it is best to attack with the griffins.

The second scenario is similar to the first one, except that now team A have knights and paladins but neither archers nor rangers; team B controls only paladins.

The main difference between the first scenario and the second one is that paladins cannot attack griffins, while rangers can. Therefore in the first scenario the Environmental Advantage explanation has a small chance to fail, while in the second it can only fail in case of a timeout, which happens when the griffin does not kill the paladin in a reasonable amount of time.

Finally, the third scenario is the same as the second one, except that we switch the order of the first and second part.

The greedy heuristic is very effective in the first part of each scenario, since the kinds of events in these scenarios are deterministic and the discrepancies have one fixed explanation. When we switch to the second part of each scenario, though, the greedy heuristic will take a considerable amount of time to react to the sudden change. Stochastic explanation, instead, will react faster since it will pick a different explanation to test sooner and realize it is now working. Random explanations will have an unpredictable behavior and we consider it only for comparison reasons.

The charts will present the number of iterations on the x-axis and the number of failed explanations on the y-axis. We consider as failed those explanations that provided a negative feedback when their reaction was executed. Part one of each experiment is run for 50 iterations; part two is then run for another 50 iterations for a total of 100 iterations per experiment. Each experiment is executed five times, and the results are finally averaged.
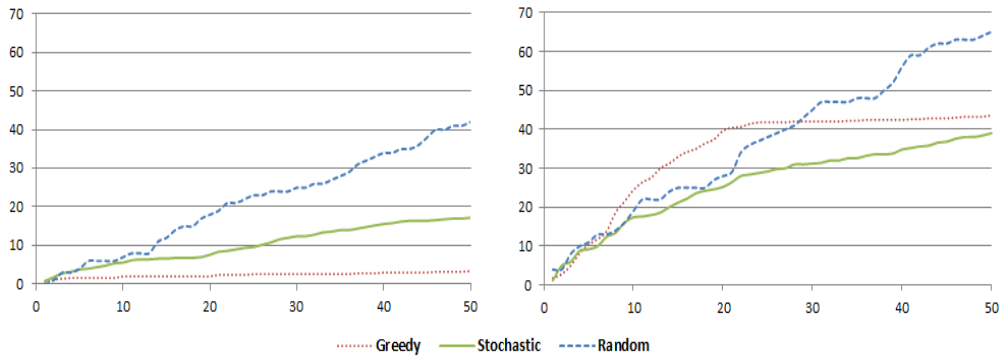
6.2 Results

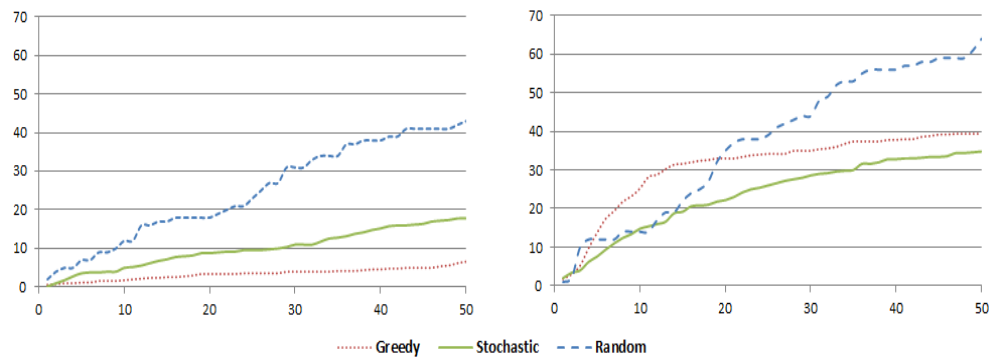**Figure 7: Results for the first scenario for parts 1 (left) and 2 (right)**

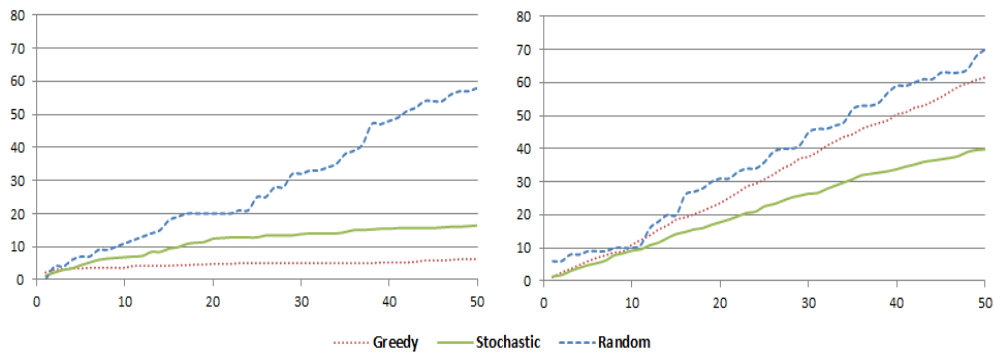**Figure 8: Results for the second scenario for parts 1 (left) and 2 (right)**

Figure 9: Results for the third scenario for parts 1 (left) and 2 (right)

The results are consistent with our expectations: in all three scenarios, in part-1 Greedy Explanations do outperform Stochastic Explanations: the former caps between 3 and 8 failed explanations, while the latter caps at approximately 18. However, when the scenario switches to part-2 (stochastic event), Stochastic Explanation react much quicker to the new conditions of the environment. In fact, we can see how in all three scenarios, the number of failed explanations in part-2 for Stochastic explanations caps at around 40, while Greedy explanations on average cap at 50, and never below 40. Random explanations as expected present an unpredictable behavior and, since there is no learning involved at all, the curve keeps climbing for all three scenarios and in both parts.

These results show how an agent that uses Stochastic explanations can and will react faster to unexpected events than one that relies on deterministic explanations. A deterministic agent relies entirely on its past experience to explain anomalies, and this works perfectly as long as this experience continues to be true under the current conditions of the environment. Once the environment changes, as it often happens when we deal with stochastic domains, a deterministic agent does not have the ability to question its perception of the environment and therefore react in a timely manner, while a stochastic agent does it on a regular basis. This results in a much faster reaction time to unpredicted events. To decide if we should use a deterministic approach or a stochastic one, we should first inspect the environment's nature. If unpredictable events can happen frequently, then a stochastic approach will guarantee faster reaction times to these events; if, instead, the environment seems to present a constant behavior, or its state mutates very rarely, then a

deterministic approach will provide a lower number of failed explanations. Regarding our experiments, we believe that stochastic explanations are a better fit to an highly unpredictable domain such as RTS games.

We ran a second set of experiments to show how GENERATE_GOAL+ (Section 4.7) can exploit the knowledge learned from past mistakes to dramatically reduce the number of discrepancies the agent will run into. The experimental setup is the same as the previous experiment; both agents use Stochastic explanations: the first, implements the *error learning* algorithm (i.e. GENERATE_GOAL+), while the other uses the normal goal generation code (i.e. EXP_GEN).
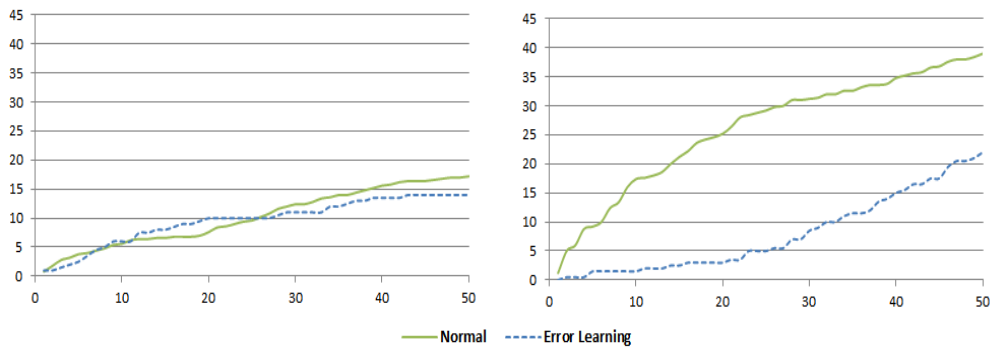

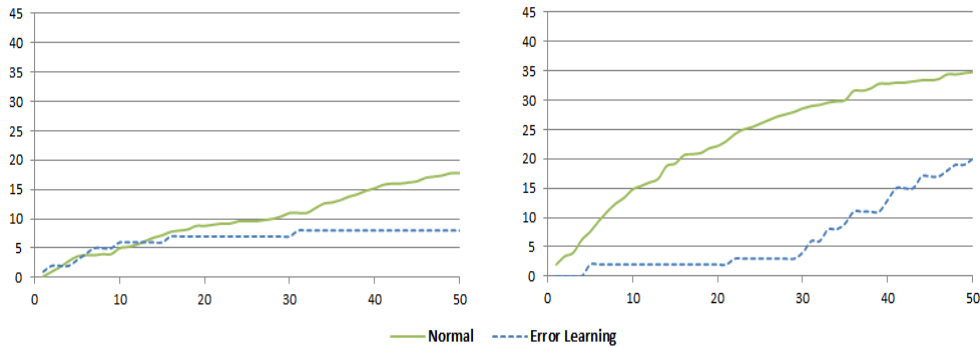**Figure 10:Results with learning for the first scenario for parts 1 (left) and 2 (right)**

Figure 11: Results with learning for the second scenario for parts 1 (left) and 2 (right)
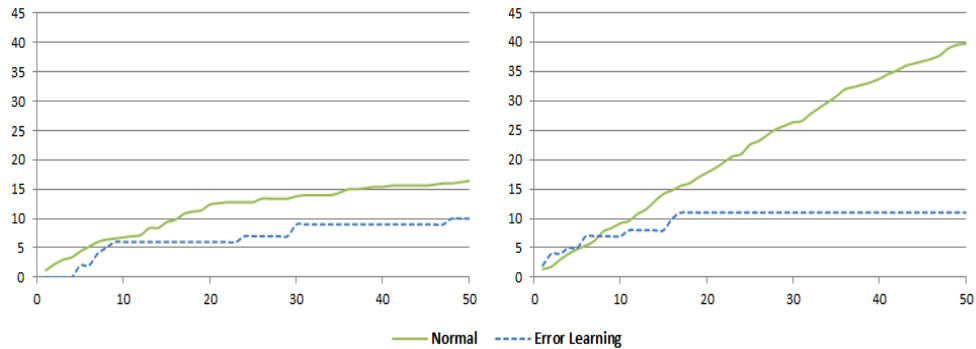
Figure 12: Results with learning for the third scenario for parts 1 (left) and 2 (right)

The results show how making use of the knowledge acquired from past mistakes can dramatically increase the performance of an intelligent agent. In all three scenarios and both in part-1 and part-2 for each scenario, the Error Learning agent that uses GENERATE_GOAL+ clearly and consistently outperforms the agent that uses the EXP_GEN algorithm. This behavior is even more evident in part-2 of each scenario, where the agent is presented with an unexpected event, making the reaction to the event even faster.

6.3 Experimental Setup: CombatPlan

Finally, we present an experiment that uses the CombatPlan algorithm. As discussed in Section 4.8, the agent implementing the CombatPlan algorithm makes use of the knowledge acquired during its training phase to generate a combat plan: a target assignment for each of the agent's units. The combat plan is then executed and every unit attacks simultaneously; this situation is very similar to the battles that happen in a real game of Wargus.

58

The experiment consists of two scenarios, one used for training and the other one used for battle; in both scenarios the agent and the enemy AI have a fixed number of units. The first scenario is similar to the scenarios we showed in the previous Sections, and it is only used for training. During the execution of this scenario, the agent uses the EXP_GEN algorithm to acquire as much knowledge as possible about the discrepancies that can occur in the environment and their respective explanations. In the second scenario, instead, the agent uses the knowledge acquired from the mistakes made during training to craft a combat plan and then executes it; in this phase, there is no learning involved. The enemy units are going to be close to each other and they will all participate in combat at the same time, rather than having a lot of one-on-one combats. To create a slight bias, the agent will have a unit more than the enemy AI. This justifies the claim that the agent should always win if the combat plan is crafted wisely. If the plan is poorly designed, instead, the units will attack targets they are weak against, and even if the agent has a unit more than the opponent, it will lose the battle. The scenario is run using five different numbers of training rounds, or in other words five different training levels. These are: *No training, 10, 20, 30,* and *50* training rounds.

The chart's x-axis shows the number of iterations, while the y-axis shows the cumulative difference in score between the agent and the opponent. Each training level runs for 50 iterations, and the results are averaged on 5 executions, for a total of 250 iterations per training level.
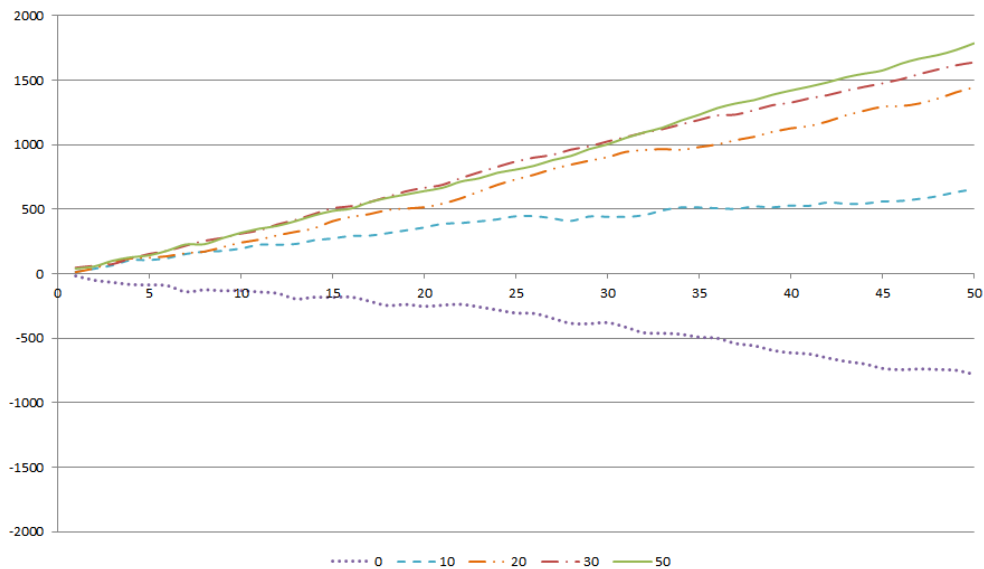
6.4 Results

Figure 13: Combat Plan results showing the effects of incremental amounts of training

The results clearly show the effect of training on the performance of the combat plan. The agent trying to create a combat plan performing no training at all presents a consistently negative score difference from the enemy AI. This is because the target that each of the agent's units gets assigned to has a high probability of being a bad choice. A short training, like 10 rounds, already lets the agent make a more informed choice for deciding a target for each unit. This leads to a positive cumulative score difference. 20 rounds of training provides a huge performance improvement from 10 rounds of training: now the agent picks almost always the best unit to attack and the score difference is noticeable. 30 rounds of training still provide a performance increase, but less evident. Finally, 50 rounds of training still

improve over 30, but very slightly. This tells us that we are approaching the optimal combat plan, and each unit is assigned to attack the enemy unit it is strongest against.

This experiment confirms our claim regarding the importance of learning from past mistakes. An agent that is able to question its own cognitive process, and that is able to treasure the discrepancies it runs into and convert them into knowledge about the environment can then utilize this knowledge in the future to make better choices about its own actions.

Chapter 7   Conclusions

7.1 Future Work

Our experiments have a common characteristic: the set of possible explanations to a given discrepancy was predetermined and did not change over time. Learning this set is an extremely challenging and fascinating problem that we left for future work. In this Section we want to discuss this issue further and possible approaches to solve it.

As we mentioned in Section 4.2, an explanation is just a label, and it does not necessarily have to be descriptive of the discrepancy it is referred to. Learning the reaction associated with an explanation is the hardest challenge, and it involves learning a policy that fits the current discrepancy. We could theoretically have a large set of policies and we could test each one of them until we reached a satisfactory state. This policy would then be picked as the reaction and we would have learned a new explanation. However, we would still have a fixed set of policies. We could nonetheless enrich this set by combining different policies to create new, more complex ones that were not present in the original set.

One must consider the cost of this operation. Learning new policies this way would have a huge computational cost, and the required training for the agent would be extremely time-consuming. To improve, we might proceed in the following way: we start the execution of the agent with some predetermined, engineered explanations and their respective reactions. Whenever a discrepancy is detected, we

test each explanation, observing the outcome. When *none* of the explanations seems to work, it means that our set is not complete and we should add a new explanation. To do this, we will use the mechanism explained earlier: we will try different policies, even combining them, until the environment provides a positive feedback. We will then pick that policy (or set of policies) as the reaction for our new explanation, and we will add it to the set of possible explanations.

Every time a new discrepancy is detected, now the set of possible explanations is larger and learning the correct probability distribution will require more training and therefore more time. Also, the stochastic explanations learned prior to the inclusion of the new explanation to the set might need to be adjusted, because even if we learned a satisfactory explanation in the past, it might be possible that a better one exists and can be discovered in the future.

There are still many open problems, and probably the biggest one is to find how we can combine policies efficiently. If done naively, this operation will require too much time to be practicable. We need an algorithm that takes into consideration different factors and that is able to reason on policies, making intelligent choices regarding the policies it picks and the state of the environment it is trying to explain.

7.2 Closing Remarks

In this thesis we presented a novel approach to explaining discrepancies using Stochastic Explanations; to do this, we used a synergy of RL and CBR mechanisms. The EXP_GEN agent uses Stochastic Explanations to explain anomalies, thus increasing its knowledge of the environment. In our results we

showed how this process enables the agent to react rapidly to unpredicted events, outperforming an agent that uses a greedy heuristic to pick an explanation.

The knowledge acquired explaining anomalies can be reused by the agent to decide what to do next. This way, the agent will be less likely to incur into discrepancies in the future. We showed how an agent that performs this operation using the GENERATE_GOAL+ algorithm outperforms an agent that only uses EXP_GEN. This fact introduces the notion of *learning from mistakes*. The more mistakes an agent incurs during its training phase, the less the mistakes the agent will make after. Explaining anomalies therefore is not an isolated activity, but the explanations are used continuously by the system throughout its execution to make better choices.

This concept is shown again in the last of our experiments with the CombatPlan algorithm. Here we showed how the amount of training the agent performs influences the agent's performance when tested in a battle. The reason for this is because the agent, during training. makes mistakes and tries to explain them. Learning the correct probability distribution of each possible discrepancy takes time and if we don't provide enough training, the agent will not be able to make good choices during the combat phase.

Explaining anomalies in a stochastic domain is a complex, fascinating task. AI strives to mimic and outperform human behavior, and explaining the world is something humans do, or try to do, since they are born. This is why the intelligent agents we write in the future should have the ability to be surprised, and to try to

explain unexpected events in order to fill the gap between the humans' and the machines' interpretation of the world.

Bibliography

Cox, M. T. (1996). Introspective Multistrategy Learning: Constructing a Learning Strategy under Reasoning Failure. *Doctoral Dissertation, George Institute of Technology.*

Cox, M. T. (2007). Perpetual self-aware cognitive agents. *AI Magazine 28(1).*

Jaidee, U., & Muñoz-Avila, H. (2012). CLASSQ-L: A Q-Learning Algorithm for Adversarial Real-Time Strategy Games. *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference.*

Klenk, M., Molineaux, M., & Aha, D. W. (2012). Goal-driven autonomy for responding to unexpected events in strategy simulations. *Computational Intelligence*, In Press.

Molineaux, M., Aha, D. W., & Kuter, U. (2011). Learning Event Models that Explain Anomalies. *In T. Roth-Berghofer, N. Tintarev, & D.B. Leake (Eds.) Explanation-Aware Computing: Papers from the IJCAI Workshop.* Barcelona, Spain.

Molineaux, M., Kuter, U., & Klenk, M. (2012). DiscoverHistory: Understanding the Past in Planning and Execution. *Proceedings of the Eleventh International Conference on Autonomous Agents and Multi-Agent Systems.* Valencia, Spain.

Nau, D. S., Au, T.-C., Ilghami, O., Kuter, U., Murdock, W. J., Wu, D., et al. (2003). SHOP2: An HTN planning system. *JAIR*, 379-404.

Ricci, F., & Avesani, P. (1995). Learning a local similarity metric for case-based reasoning. *International Conference on Case-Based Reasoning (ICCBR-95)* (pp. 301-312). Sesimbra, Portugal: Springer.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning: An introduction.* Cambridge, MA: MIT press.

Wilke, W., Lenz, M., & Wess, S. (1998). *Case-based reasoning technology, from foundations to applications.* Springer-Verlag.

Vita

Giulio Finestrali was born in Carpi, in the north of Italy, on September 9th 1988. His father is Andrea Finestrali and his mother is Eva Lepore. In September 2007 he began his undergraduate studies in Computer Engineering at Modena and Reggio Emilia University (Modena, Italy), from which he graduated in April 2011. In January 2012 he joined the Master's degree program in Computer Science at Lehigh University. He is expected to graduate in May 2013.