Æip: A Generalized Framework for the Study of Interactive Learning

by

Denis V. Batalov

A Thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

School of Computer Science
Carleton University
Ottawa, Ontario, Canada
January 2007

Your file Votre référence
ISBN: 978-0-494-23286-6
Our file Notre référence
ISBN: 978-0-494-23286-6

Abstract

The general subject matter of this thesis pertains to the study of the framework within which one can construct, and experiment with intelligent adaptive systems. One of the important contributions of this thesis is the generalization of the feedback loops, currently studied in the fields of Reinforcement Learning and Learning Automata, to allow for a wide variety of experiments, including multi-agent and multi-environment interactions. This generalization naturally led to the investigation of learning to play games under extreme conditions of minimal *a priori* information. As another outcome of the above generalization, we consider alternatives to the reward/punishment signals for supplying the goal information to the artificial systems. This work has resulted in other important contributions — the classification of goals according to their *arity*, and the subsequent study of the novel *feedback* signal as an alternative to reinforcement. A learning task can now be analyzed and classified based on its arity. The introduction of the feedback signal has further led to the development of the discretized forms of learning algorithms and to DQ-learning in particular. Knowledge of the arity of a task allows us to construct the corresponding feedback signal, and, hence, to apply the DQ-learning algorithm with *guaranteed convergence* and memory savings which enable us to tackle more challenging tasks. Given that these ideas were borne out of our generalization of the agent-environment interaction, we believe that we have shown this generalization to be fruitful. In particular, we have demonstrated its theoretical merit and its applicability in an ensemble of areas including game playing, organization of data in adaptive lists, and differentiated robot control. We further believe that other important contributions can result from this work.

Dedicated to the Power of Love and all the people in my life whose love is an unfaltering source of strength I draw from in all my endeavours. All you need is love!

♡

Acknowledgements

I would like to express my deep-felt gratitude to my thesis supervisor, Prof. B. John Oommen. Over the past years, he has been a teacher, a colleague, and a true friend whose encouragement and support I could always rely on. I am constantly inspired by his energy, dedication, and generosity.

I would also like to thank many current and former faculty members and support staff from Carleton University and University of Ottawa. They taught me and helped me so much over the years.

I am grateful to the following organizations for their financial support during my studies: Natural Sciences and Engineering Research Council (NSERC), Ontario Graduate Scholarship Program (OGS), Nortel Networks Corporation, David and Rachel Epstein Foundation, and both University of Ottawa and Carleton University.

Finally, I would like to acknowledge the secret co-authors of this thesis — my beloved parents Valerii and Galina. They opened my eyes to this wondrous world, taught me to appreciate its mysteries, and showed the horizons of endless possibilities. I am but a mere extension of their sense of adventure, their creativity, and most importantly, their love. Thank you with all my heart!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The work described in this thesis was, in part, motivated by our desire to study, experiment with, and understand "intelligent" systems. Precisely what constiutes such a system is still a matter of debate, and so we would like to preface the rest of this introduction with a quote by the renowned psychologist Robert Sternberg [80, p.8]:

> "Many theorists of intelligence would define the locus of intelligence as occurring neither wholly within the individual nor wholly within the environment, but rather within the interaction between the two ... Thus it may be difficult to understand intelligence fully without first considering the interaction of the person with one or more environments ..."

We expand on our own outlook on the philosophy of "intelligence" in Appendix A where we outline what we believe is a promising approach to constructing the theory of intelligence. While this thesis does not offer such a theory, it provides a step in this direction — a framework within which such theories may be developed, prototyped, and hopefully tested.

Intelligent behaviour, we believe, is necessarily goal-oriented and must involve the interdependent abilities to learn (acquire the rules of the environment) and plan (apply these rules to reach its objectives). We also believe that any successful theory of intelligence must

be based on the observable behaviours of the systems to which such intelligence is attributed. Since behaviour occurs in the context of interaction with the surrounding environment, intelligent behaviour must be studied in exactly the same setting. This is precisely the inspiration for our interest in *Learning by Interaction* which led to the development of the central building block of this thesis — the generalization of the Agent-Environment interaction protocol (see Chapter 3) and all our subsequent results.

1.1 Learning by Interaction

Driven by our interest in Reinforcement Learning, we experimented with a variety of adaptive algorithms. During the course of setting up these experiments and implementing necessary algorithms, environments and controllers in a non-systematic fashion, it became apparent that a lot of effort could be saved if all the modules and packages were built on the foundation of a common experimentation platform. Such a platform would tremendously simplify empirical studies by allowing for standard interchangeable components, i.e. agents and environments. It also became clear that it would be much simpler to compare characteristics of individual algorithms if they were interacting with exactly the same environment.

Motivated by this vision we proceeded to examine various frameworks and experimentation scenarios currently used in the fields of Reinforcement Learning (RL) and Learning Automata (LA). The literature systematically describes the abstraction as a single agent (automaton) interacting with a single environment (see Section 2.4.1). In practice, however, agents and environments may consist of multiple interacting components (as described in the Section 2.4.2.6 on LA). Multi-agent interactions were also studied where the distinction between an agent and an environment is eroded since one agent becomes *part of* the environment for some other agent. This led us to formulating the goal of constructing a single Interaction Protocol and a single framework within which all such experiments can be easily conducted [11].

It becomes clear that as the Investigator constructs such a framework, he has to reconsider the fundamental assumptions of the existing frameworks. One such assumption is the perception of reinforcement as the best (or, in some cases, as the only) way of training the agent to reach its goal. One of the important contributions of this work is the study of an alternative way of supplying such goal information and the notion of goal analysis. This, in turn, naturally led us to an idea of an explicit discretization of Q-learning, not unlike the discretization of the (linear) reward-inaction scheme in the field of LA (see Section 2.4.2.5). While such discretization was necessarily implicit in the computer implementation of learning algorithms, we propose to explicitly relate the goal of the agent and the degree of the resulting discretization through arity-based goal analysis.

If the reinforcement signal is replaced by a signal of a different sort (*feedback* signal in our case), it is no longer appropriate to consider the corresponding algorithm as being a *Reinforcement* Learning algorithm. This motivated us to suggest a more general framework, namely, Learning by Interaction or Interactive Learning. This naturally encompasses the sister field of LA as well.

1.2 Organization and Contributions of the Thesis

We outline here the structure of our thesis and list what, we believe, are its main contributions. In Chapter 2, we conduct a survey of the various subfields in the domain of AI that, we feel, are relevant to the construction of intelligent systems related to our philosophy of AI. We focus our attention on the fields of RL and LA, where the learning and planning aspects of "intelligence" converge.

1.2.1 Generalization of the Feedback Loop

Chapter 3 discusses the first important contribution of this thesis. It is the generalization of the feedback loops currently studied in the fields of RL and LA to permit a wide variety of experiments. Instead of dealing with agents and environments as essentially different kinds of "things", we define a framework (called Æip) within which we can set up multi-agent and multi-environment experiments. In Æip, agents and environments are treated as being complementary, and yet as being similar *entities*. Based on this generalization, we also constructed an experimentation tool (called JAGUAR, or Java AGents Unified ARchitecture) where libraries of agent and environment algorithms are accumulated as implementations of general and interchangeable entities. This tool has a graphical user interface that allows a researcher to set up arbitrarily complex experiments by instantiating a number of entity classes, connecting the resultant entities together into an interacting system, specifying the goals to be reached by the agent entities, and observing the resultant simulated interaction. The benefits of such a tool include easier experimentation and a more objective comparison of algorithms in terms of their "intelligence". All experiments reported in the following chapters were conducted within the Æip infrastructure inside JAGUAR. Appendix D contains a brief description of this tool.

1.2.2 Eliminating Asymmetry between Agents and Environments

In the classical RL loop, the environments communicate their current *state*, while agents communicate their chosen *action*. This semantic difference presents a problem if one wants to treat agents and environments as similar entities with just "inputs" and "outputs". We, therefore, propose to replace agent actions with agent states. Such a replacement is related to the notion of *idempotency* in messaging protocols (see [30] for example). An action is, thus, a perceived transition in the "observable" state of the agent.

1.2.2.1 Arity-based Goal Analysis

Another way in which environments are different from agents is that they emit an additional real-valued reinforcement signal, as opposed to other typical signals (often discrete-valued) that deliver the current observable state of the environment. To remove this source of asymmetry between agents and environments, we proceeded to consider alternative mechanisms to the reinforcement signals, which are capable of supplying the goal information to the agent entities. This work resulted in another important contribution — the classification of goals according to their *arity*, and the subsequent study of the novel *feedback* signal as an alternative to reinforcement [12]. In this setting, we examined in great detail the question of what constitutes a "goal" in goal-directed intelligent behaviour (see Chapter 3). In addition to classifying a learning task as being either Markovian or not, we can now speak of binary or, in general, N-ary tasks.

1.2.2.2 Feedback Signal

The arity-based goal analysis allows us to construct an alternative way of supplying goal information to the learning agents — the feedback signal. We will show that by its definition, the feedback signal has attractive properties which we exploit by proving convergence of an algorithm that uses the new feedback signal. This proof takes advantage of the key insight into the nature of the feedback signal: a goal supplied in the form of the feedback signal is the equivalent of the admissible heuristic in A* search algorithm. More specifically, a version of our discretized Q-learning algorithm is guaranteed to converge to a solution in a finite number of steps if the goal is communicated in the form of the feedback signal. The said proof is an important theoretical contribution of this thesis and the gained insight into the close relationship between feedback-based Interaction Learning and the A*-based search might be the single most important contribution of this thesis.

1.2.3 Learning to Play Games without Knowing the Rules

If agent entities are to be constructed in a general way to cope with various experimentation scenarios, they must not depend on the particulars of any given environment. While we still want to be able to configure such an entity to best suit the current experiment, we should strive to build entities that know as little as possible about the other entities with which they will be interacting. This naturally led us to the investigation of agents learning to play games under extreme conditions of minimal *a priori* information. Chapter 4 (also see [13]) describes a particular set of experiments where the agent entity, in addition to having been unaware of the rules of the game to be played, was also unaware of the very existence of its opponent, and that the moves of the two players must alternate. Such a work is novel to the field of game playing.

1.2.4 Discretizing Q-learning

The introduction of the feedback signal led us to the development of the discretized forms of learning algorithms and to DQ-learning, in particular. In the Reinforcement Learning literature, little consideration is given to the fact that, in any particular implementation of a learning algorithm, the real-valued weights, used in representing the current "knowledge" of the agent, can only be stored in variables of limited floating-point precision. This limits the abilities of the agent to discriminate the many alternative solutions to a given task. We will demonstrate with our Discretized Q-learning algorithm, that we can be much more explicit about the relationship between the precision used and the kind of tasks that the agent will be able to learn. This is another important contribution of our work, and is described in detail in Chapter 4.

Knowledge of the arity of a task and the resulting discrete representation can also allow us to save memory and, hence, enable us to tackle more challenging tasks. Furthermore, specialized algorithms can be created for tasks of a given arity, e.g. for binary goals.

1.2.4.1 Solving LightsOut Puzzle

We will show that binary Markov tasks can be solved with a variant of Q-learning where only a single bit is required per table entry. Taking advantage of the memory savings resulting from the thrifty tabular representation of DQ-learning, we conducted a series of experiments with several variants of the LightsOut puzzle for goals of various arity. We demonstrate in Chapter 4 that DQ-learning makes a difference between being able to and not being able to solve the puzzle given the practical limitations on the available RAM.

1.2.5 Novel Experiments Using Æip

The Æip framework was designed to be applicable to a wide variety of experiments which involve Learning by Interaction. In particular, we considered the potential of setting up experiments in which agents either cooperated or competed with each other. The rest of the thesis describes experiments within purposefully different domains. This was done to demonstrate the wide applicability of the proposed framework. The experiments themselves, however, are also novel in the respective domains; they contribute to the understanding of the learning schemes used. It is worth noting that these novel experiments also demonstrate how multi-agent interaction can be organized within the Æip framework.

1.2.5.1 Comparison of List Organizing Strategies

Chapter 5 introduces the field of Adaptive Data Structures and focuses on the problem of the dynamic reordering of a linked list based on the empirical distributions of queries. We first discuss many of the well-known list organizing strategies, and then proceed to demonstrate how experiments can be set up within the Æip framework to compare the properties of these schemes in the so-called traditional setting. We then propose a way of comparing these strategies in a *competitive* setting, and propose ways of making such a competition "fair", as well as, examine composite strategies and their performance.

In the spirit of Section 4.4, we consider the scenarios in which the competing strategies are unaware of each other. To show the value of explicit information about the opponent, we also conducted experiments where one scheme was able to observe the actions of the other. To the best of our knowledge, this is a pioneering work, as the comparison of list organizing strategies in such competitive settings has never been done before. This is the most important contribution of Chapter 5. Such a comparison not only promotes further understanding of the schemes, but also hints at a methodology that may be useful in comparing algorithms (especially adaptive algorithms) in general.

1.2.5.2 Differentiated Robot Control

Finally, Chapter 6 presents our experiments with robot control. The novel aspect of these multi-agent experiments is the simultaneous control of the single robot by two separate and largely independent agents, hence *differentiated* control. In contrast to Chapter 5, the agents were not competing (by having opposing incompatible goals) — but "cooperated" with each other (by having the same overall objective). The differentiated nature of the control is achieved by having the agents control independent aspects of the robot, namely, the wheel position (north or west) and direction of rotation (forward or backward). The agents could act simultaneously and receive the same exact feedback from the environment.

We discuss experiments where the agents were unaware of each other, and yet still managed to move the robot into the goal cell of the Grid World, albeit in a suboptimal number of steps. This, in itself, is a testament to the robustness of the Q-learning family of algorithms. We also describe experiments where the agents acted one after another, the last agent being able to "observe" the decision of the previous, before committing to its own action. A pair of "communicating" agents is shown to be capable of converging to the optimum as the Markovian nature of the interaction is now restored. The main contribution of this Chapter is the demonstration that convergence is possible in the case of differentiated control even

when the agents are agnostic of each other. At the same time, we clearly show that with the introduction of minimal "cross-agent" communication, the convergence is qualitatively improved. Lastly, we also modify the task somewhat and demonstrate how DQ-learning and arity based goal analysis can be applied to the scenario of communicating agents.

Chapter 2

Survey of Relevant AI Research

Introduction

In the early days of Artificial Intelligence research, a lot of progress was made in a relatively short span of time which caused most scientists to underestimate the difficulty of building *intelligent* systems. In the 1960's, the general feeling was that it would be but a matter of years before we could build systems that were capable of exhibiting most of the intelligent behavioural patterns of human beings. This belief was a source of great disappointment as years passed without many major breakthroughs. The backlash was so great, that even today, we have AI philosophers (see [63]) who argue the *theoretical impossibility* of computer programs, written for the *von Neumann* architecture, to ever appear intelligent. As a result, AI researchers settled on less ambitious goals of solving individual problems that normally require human intelligence, and in a sense, abandoning, for the time being, the dream of building a single all-intelligent system capable of solving any problems requiring "intelligence". For example, programs created by the Natural Language Processing (NLP) research may be good at understanding messages communicated in human languages, yet NLP systems would not be able to play chess or prove theorems. Likewise, Machine Learning

systems designed to identify oil spills from aerial photographs cannot understand language, or plan the motion of a car. The hope is that insights, gained in each of the many subfields of AI, will allow us eventually to assemble a single system that exhibits "intelligence". To exacerbate the problem, little progress is evident even in the most fundamental area of AI — the understanding, from a formal point of view, of what intelligence is.

This sentiment is echoed by a recent article by Nils Nilsson [54] which appeared in the 25th AAAI anniversary issue of the AI magazine. Inspired by a much earlier famous article by Alan Turing [89], Nilsson gives an "onion" analogy for the development of the intelligence of a human adult:

> *At the core is a set of very sophisticated and powerful, let us say hard-wired, basic mechanisms provided by our evolutionary heritage. This core is able to build the next layer — providing the core is immersed in an appropriate environment. Together, they build the next layer and so on. If building "outer-layer" systems directly proves too difficult, perhaps, building the core will be easier.*

This emphasis on the "core" of the intelligent system also goes back to the same paper by Alan Turing [89] where he proposed the so called "child project":

> *Instead of trying to produce a programme to simulate the adult mind, why not rather try to produce one which simulates the child's?*

It may indeed be very difficult to build an entire "outer layer". This is exemplified by the slower-than-expected progress of AI research. Nilsson offers a well-known example of an essentially failed attempt which concentrated on the "outer-layer", namely, the CYC project [43]. Perhaps it is not surprising that a lot of modern AI research is concentrated on small portions of such "outer layer", abandoning for the time being, the dream of strong AI in favour of achievable progress.

This chapter attempts to review AI research from the point of view of building a "core" intelligent system — the same point of view we take in the work presented in this thesis.

2.1 AI Research Overview

We categorize AI research based on two capabilities that are widely recognized as being key to the construction of an intelligent system. Short of giving a formal definition of intelligence, it can be argued that a system exhibiting intelligent behaviour consists of two parallel and interdependent processes: *learning* and *planning*. Learning refers to an inductive process of acquiring the rules that govern the environment in which the system exists, while planning refers to the related deductive process of taking advantage of the learned rules to achieve a goal that the system must have. It is not surprising that these capabilities of learning and planning are also featured prominently in the already quoted article by Nilsson. Where appropriate we provide relevant quotes from this article.

Albeit related, the abilities to learn and plan are distinct. In itself, the acquired knowledge about the environment is passive, which means that, in addition to possessing said knowledge, a system must also know how to use it effectively to reach its objectives. For instance, we know precisely all the rules that govern a chess play or the rotations of a Rubik's Cube; yet even such specific knowledge is not sufficient to win chess games or solve the puzzle. Likewise, planning alone is not going to be effective if more accurate knowledge of the environment is not learned. For example, with the invention of the airplane, we can transport ourselves over much longer distances in much less time than was previously possible, regardless of how good our past planning abilities might have been.

Albeit distinct, the abilities to learn and plan are interdependent. Learning is achieved by exploration of the environment, and any such exploration must first be planned. Likewise, any plan should incorporate the latest acquired knowledge.

AI research can be roughly classified into emphasis on learning (as in *Machine Learning*), emphasis on planning (as in *Search* or *Planning*), and into mixed approaches (as in *Reinforcement Learning*). These approaches are examined in more detail below. Since we concentrate mainly on the latter mixed approaches in our thesis, we attempt to present a

more detailed account of that very research. The overview presented here is by no means complete. It is intended to be representative of the state of the art, and of the various possible avenues.

2.2 Acquiring Knowledge — Learning

A definition of Learning suggested by Simon (from [29]) is:

> ***Definition***: Learning denotes changes in the system that are adaptive in the sense that they enable the system to do the same task or tasks drawn from the same population more efficiently and more effectively the next time.

This definition underscores changes in the system (possibly invisible to the outside observer) that are effected by learning. The learned knowledge *enables* the system to perform better, but does not necessarily entail an improvement.
A more recent and slightly more formal definition of learning appears in [50]:

> ***Definition***: A computer program is said to **learn** from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

This definition of learning is slightly different from the previous one because it also incorporates planning ability. According to this definition the only way learning can occur is if we could register an improvement in performance. It thus tries to quantify learning with an objective procedure. It is our opinion that such a definition is better suited for intelligence itself as a measurable property of the system, while the ability to learn and plan can only be quantified if the innerworkings of a system are known. This position, however, is arguable.

2.2.1 Supervised Learning

Supervised Learning is characterized by systems learning from a teacher, who is usually assumed to have full (possibly noisy) knowledge of the subject to be learned. Most of the research deals with variations of the concept classification problem of labeling an instance of some pattern with the correct class. In its most general setting, it incorporates well known approaches of *Version Spaces*, *Decision Trees*, *Instance Based Learning*, and *Bayesian Learning*. Several *Artificial Neural Network*[1] configurations are also geared towards solving such problems.

Concept Learning is the task of acquiring an operational[2] definition of a general category (or *target concept*) given a sample of positive and negative training examples of the category. Such a definition is often called a *hypothesis* which is described in some language of constraints. Concept learning can, thus, be viewed as a problem of searching through a predefined space of potential hypotheses for the hypothesis (or hypotheses) that best fits the training data. In the remainder of this Section our review essentially follows [50].

2.2.1.1 Version Spaces

It turns out that the hypothesis space has a naturally occurring structure — a general-to-specific ordering of hypotheses. *Version spaces* and the related *Candidate-Elimination algorithm* described in this section takes advantage of this ordering. They were first introduced by Mitchell, and a more detailed account can be found in [49] and [50, ch.2].

A version space is a set of all hypotheses that are consistent with the observed training examples. It contains all plausible versions of the target concept, and is the output of the Candidate-Elimination algorithm. The algorithm performs a bidirectional search in the space

[1] ANNs can also solve *Unsupervised* and *Reinforcement Learning* tasks some of which are described later.

[2] Systems that have acquired such a definition should be able to categorize an instance, but do not have to represent a definition in some explicit or constructive form.

of hypotheses from the two starting boundary sets: the most general (denoted G_0) and the most specific (denoted S_0). These boundary sets enclose the current version space based on the general-to-specific ordering of hypotheses.

Given a set of positive and negative training examples and a hypothesis representation language, the concept learning problem can be now reformulated as a search for a single hypothesis that accepts all positive examples and rejects all negative ones[3], in other words, fitting the training data. Why would a hypothesis fitting the training data have any predictive power over the unseen examples? It turns out that this is the fundamental assumption of inductive learning. Quoting from [50, p.23]:

> "**The inductive learning hypothesis.** Any hypothesis found to approximate the target function well over a sufficiently large set of training examples will also approximate the target function well over other unobserved examples."

The above statement is purposefully imprecise when it talks about "good" approximations. Having too "good" of an approximation can be detrimental to the predictive properties of the learned hypothesis due to the problem of *overfitting*, which will be further discussed in the context of Decision Trees in the next section.

Let us now return to the Candidate-Elimination algorithm. As was said earlier, it is based on the general-to-specific ordering of hypotheses. This general-to-specific ordering is only a partial order relationship because there exist pairs of hypotheses such that their corresponding sets of positive instances intersect. This is essentially the reason why the version space is specified by boundary *sets* rather than single most specific and most general consistent hypotheses.

According to the algorithm, the boundary sets are iteratively adjusted every time a new training example is introduced. This produces a sequence of pairs $\{(G_i, S_i)\}$ which

[3]We are assuming, of course, that there are no errors (or noise) in the training data.

eventually either converge to a single consistent hypothesis (the target concept is found), or we run out of training examples (the last version space is a set of plausible hypotheses), or the version space is reduced to an empty set (in which case a consistent hypothesis in the current representation does not exist). The last possibility could be due to the poor choice of representation, which happens, for example, when a single conjunction of attributes is not sufficient, and we need to allow for disjunctions as well. Another reason could be the errors in the training examples themselves. The original Version Spaces approach is known for its inability to cope with noisy data. Is there a single powerful representation language capable of expressing any possible hypothesis?

The answer to this natural question is, in our opinion, one of the fundamental truths in inductive learning. Concisely stated, *the learner, that makes no* a priori *assumptions regarding the nature of the target concept, has no rational basis for classifying any of the instances that were not yet seen.* This means that the inductive learning hypothesis no longer applies. This topic warrants further elaboration.

By choosing a specific representation language, we introduce a bias causing the algorithm to consider hypotheses in the chosen language only and completely ignore all others including, possibly, the target hypothesis. Such a bias, however, must be present for inductive inference to occur. This seems surprising, but it can be shown [50, pp.39–45] that if we choose the most general language of arbitrary disjunctions, conjunctions and negations over attribute vectors, the version space obtained by the Candidate-Elimination algorithm will not be able to unambiguously classify any of the unseen examples. This is because exactly half of the hypothesis will be classifying the unseen instance as positive, and another half as negative, while the boundary sets will contain only the disjunctions of the training examples.

In general, the classification decisions of the inductive learner cannot be proven to be correct. In other words, they do not follow deductively from the training data. Thus, the inductive bias is a set of assumptions that, if combined with training data, will logically *entail* the learner's classifications.

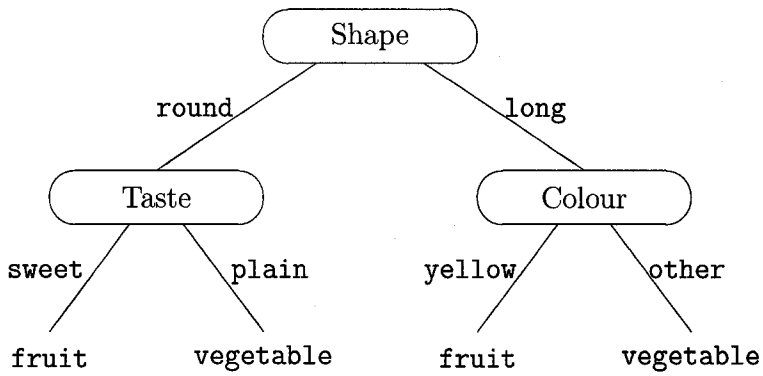

Figure 2.1: One decision tree that classifies objects as either fruits or vegetables.

2.2.1.2 Decision Trees

Decision Tree learning is another method for approximating discrete-valued target functions, which works by constructing a disjunction of conjunctions in the form of a tree. Tree learning methods are among the most practical and, hence, the most popular of inductive inference algorithms because they are simple, yet versatile enough to be applicable to a broad range of tasks. Decision tree methods have been applied with equal success to tasks such as learning to diagnose medical cases and learning to assess credit risk of loan applications. The best known decision tree learning algorithms are ID3 [65] and the subsequent C4.5 [66], both due to Quinlan.

A node in a decision tree represents a "decision" or a test of a particular attribute for some instance. The branches descending from a node represent the particular outcomes of such a test. Figure 2.1 shows an example of a decision tree for the fruit/vegetable classification task. It correctly classifies all the training examples from the table shown earlier. A new instance $potato\langle \texttt{brown}, \texttt{round}, \texttt{plain} \rangle$ will also be correctly sorted down the tree and classified as a vegetable.

In general, decision tree algorithms are best suited for problems where instances are described by a fixed set of attributes (*Colour*, *Shape*, *Taste*) and their values (`red`, `yellow`, `green`), the target function has discrete values (`fruit, vegetable`), and the training data has errors or, possibly, missing attribute values. The robustness of the tree learning methods with regard to noisy data is particularly attractive in real applications.

Unlike Candidate-Elimination (which incrementally adjusts the version space, one example at a time), ID3 considers all examples at once. It starts by selecting the root attribute of the tree by deciding which one single attribute can best classify *all* the examples. This decision is based on solid information-theoretic notions of *entropy* and *information gain*. The entropy $\mathcal{E}$ of a set of examples S with c different classes is given by:

$$\mathcal{E}(S) \equiv \sum_{i=1}^{c} -p_i \log_2 p_i,$$

where p_i is the proportion of class i examples,

$$p_i \equiv \frac{n_i}{\sum_{j=1}^{c} n_j}.$$

Having defined entropy as a measure of impurity of a set of training examples S, we can now select the root attribute based on the combined purity of the resulting subsets S_v corresponding to the particular values v of the chosen attribute A. Quite simply put, we want to select an attribute that best classifies the training examples all by itself. We can do this by introducing a notion of the *information gain* $\mathcal{G}$ defined for each attribute A as:

$$\mathcal{G}(S, A) \equiv \mathcal{E}(S) - \sum_{v \in A} \frac{|S_v|}{|S|} \mathcal{E}(S_v).$$

Gain is, thus, defined as the difference between the original entropy and the expected entropy (the weighted sum of entropies of each of the subsets S_v) after learning the value of attribute A. In other words, it is the information obtained about the value of the target function given the knowledge of the attribute. The root attribute is chosen as the one with highest information gain.

In the most general case, the algorithm will recursively construct child trees for each of the branches corresponding to each of the resulting subsets S_v. The algorithm stops either when no information gain is obtained by introducing child trees, or when all attributes are already used in the parent nodes.

ID3 can be characterized as conducting a simple-to-complex, hill-climbing search through the space of all possible decision trees. It starts with an empty tree (no attribute values are examined to produce a classification) and constructs progressively more complex trees by adding child nodes. The hill-climbing search is guided by the information gain function similar to the error function in the gradient descent of neural networks described later. This search does not consider all possible trees that are consistent with the training data (such as the tree in Figure 2.1), and unlike the Candidate-Elimination, maintains a single current hypothesis tree rather than the whole version space. It is a greedy search that never backtracks except for the special case of pruning mentioned below. The search can thus converge to a suboptimal tree that it discovers along the single search path. On the other hand, it imposes no representational bias because every finite discrete-valued function can be represented by some decision tree. The inductive bias of ID3 is, thus, purely preferential, based on the specific trees (out of all possible consistent trees) that the algorithm "prefers" to construct. The inductive bias of ID3 can be summarized as a preference for shorter trees that places highest information gain attributes closer to the root of the tree.

The rationale for preferring shorter trees is based on the famous Occam's Razor principle introduced by William of Occam around the year 1320. The principle states that "one should prefer the simplest explanation that corresponds to the observed phenomena". Restated with concept learning in mind, it instructs us to prefer the simplest hypothesis that fits the training data. The debate over the merit of this principle has been raging for centuries among scientists and philosophers, and is still without a resolution. One argument for the Occam's

Razor is that, since there are fewer simple hypotheses[4] than complex ones, it is less likely to accidentally come across a simple hypothesis that fits the data. Arguments, however, can also be put forward against such a position [50, p.65]. Nevertheless, Occam's Razor seems to enjoy tremendous popularity among machine learning researchers. It exhibits itself, for example, in the Minimum Description Length principle in Bayesian learning.

While the general bias of ID3 is towards shorter trees, it is still required to construct a tree that best fits the data. If the concept is complex (all practical applications involve only such concepts), the trees may have to grow deep to become consistent with all the examples. In such cases, the leaves of the tree may consist of sets that contain very few examples, and possibly even a single example. While such a tree will perfectly classify all the training examples, it loses its predictive power as it fails to generalize. This condition is referred to as *overfitting*, which can also occur when training data contains noise, or when there are simply too few examples to produce a representative sample of the true target function. Any solution to overfitting must reduce the depth of (i.e. *prune*) the trees being constructed.

2.2.1.3 Instance Based Learning

Instead of storing an explicit description of the target concept in some chosen representation, Instance Based Learning (IBL) methods simply store all or some of the training examples. They are also often called *lazy* learning methods because they postpone generalizing beyond these examples until a new instance has to be classified. In order to determine the class of a new instance, an IBL algorithm must examine its relationship to all the stored training examples. A representative example of an IBL method is the well-known Nearest Neighbor algorithm, where the class of a new instance depends on the proximity of the instance in the n-dimensional feature space, to the given training examples whose classes are known. An important advantage of IBL methods is that they can construct a different approximation

[4]Based on a purely combinatorial arguments.

to the target concept for each instance to be classified. They work particularly well when the class boundaries are complex and convoluted.

Since the learning phase for IBL algorithms typically consists of simply storing the training data, the classification phase bears all the computational brunt. It is also expected that IBL methods will consume more memory, since they store all the information contained within the examples, because generalization is delayed until the time of classification. Deciding which features are most relevant for determining proximity is another important consideration. If all features are treated equally, the instances that are truly most similar may appear far apart. One approach to overcome this problem is to weigh each feature differently based on how relevant the feature is thought to be. This is analogous to dilating some, and contracting other feature axes. One special case of this approach is to completely eliminate the least relevant features from the instance space. This amounts to projecting the original n-dimensional space onto a subspace with fewer dimensions.

Another IBL method is based on Locally Weighed Regression, a special case of which corresponds to local approximations with radial basis kernel functions. A global approximation to the target function is, thus, represented by a linear combination of many local kernel functions[5].

IBL approaches can work even in cases where the features of the training examples are symbolic in nature and cannot be represented as points in Euclidean space. Case-based Reasoning (CBR) [38] is a learning paradigm which, just as the k-Nearest Neighbor, defers the generalization of training examples until a new instance to be classified is observed. At that juncture it takes into consideration only a subset of the most similar known instances in classifying a new one.

[5]These kernels are often chosen to be Gaussian functions due to their attractive mathematical properties.

2.2.1.4 Other Supervised Learning Paradigms

The present survey is not meant to literally review every known technique. Consequently, we have omitted from the discussion several important ones. Specifically, we did not discuss Genetic Algorithms [25], Bayesian Learning [35], or Neural Networks [29][31]. The latter are particularly important as they underlie many AI algorithms including Concept Learning (backpropagation), Reinforcement Learning (function approximation), Unsupervised Learning (Kohonen maps), and Pattern Recognition (linear classifiers).

2.3 Applying What is Known — Planning

Earlier in this chapter we stated that any system that exhibits "intelligent" behaviour must incorporate two interdependent processes of learning and planning. This view is shared by other AI researchers, but possibly with a slightly differing terminology. Hutchinson, for example, writes [34, p.9]:

> "One naturally thinks of the solver and learner as two subroutines in symbiosis within an overall program. This situation is common, but there are cases where they are completely separate, and other cases where the learner and problem solver are the same — the learner learns how to improve its own performance."

Here, the "problem solver" is, essentially, a planning unit that deductively constructs a path (a plan) from the initial starting state. Since most planners are searchers, planning and searching can be considered synonymous in certain contexts.

2.3.1 Search

While classical search is usually a deductive activity (i.e. it cannot reveal information that does not deductively follow from the existing knowledge), surprisingly, search is also used

as a part of the inductive process. Assuming the chosen inductive bias is correct, learning is reduced to the deductive search in the hypothesis space. In the following discussion of search we will, however, ignore other applications of search, and concentrate on the classical notion of search as constructing a plan of action, or finding a path from starting state to the goal state.

Many different search algorithms exist. They range from the universally known depth- and breadth-first search techniques, to the lesser known hierarchical searches with abstractions [28] and real-time search [39]. The deductive nature of search can be best demonstrated by the classic search problems in the so-called BLOCKS world [55, pp.152–157]. In its simplest variant, the world consists of three wooden blocks labeled **A**, **B**, and **C**, and a **Table**. The objective of a planner is to find a sequence of block moves to produce the target state given the initial starting state. Figure 2.2 shows the possible start and goal states. Given

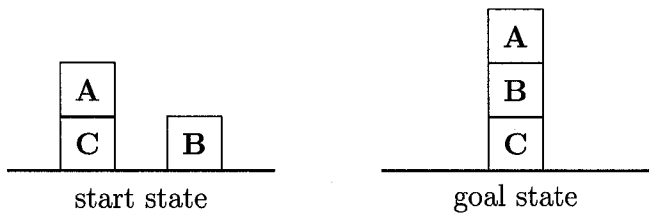

Figure 2.2: The start and goal states in the BLOCKS world

the four literal symbols and the relation $\mathrm{on}(x, y)$, we can describe any state in this world[6]. For example, the starting state can be described as

$$\mathrm{on}(\mathbf{C}, \mathbf{Table}) \quad \wedge \quad \mathrm{on}(\mathbf{A}, \mathbf{C}) \quad \wedge \quad \mathrm{on}(\mathbf{B}, \mathbf{Table}),$$

and the goal state as

$$\mathrm{on}(\mathbf{C}, \mathbf{Table}) \quad \wedge \quad \mathrm{on}(\mathbf{B}, \mathbf{C}) \quad \wedge \quad \mathrm{on}(\mathbf{A}, \mathbf{B}).$$

[6]We are assuming that whatever is not explicitly stated to be true, is considered false.

Given this language of state descriptions, we can define two operators, *stack* and *unstack* that have the following preconditions (above the line) and postconditions (below the line):

$$stack\frac{\mathbf{on(x, Table)} \wedge \neg\mathbf{on(z, y)}}{\neg\mathbf{on(x, Table)} \wedge \mathbf{on(x, y)}}, \qquad unstack\frac{\mathbf{on(x, y)}\neg\mathbf{on(x, Table)}\wedge}{\mathbf{on(x, Table)} \wedge \neg\mathbf{on(z, y)}}.$$

Note that the pre- and postconditions for the two operations are reversed. These operators define the *search space* as a labeled directed rooted graph. Figure 2.3 shows a complete expansion of the BLOCKS search space with three blocks.

In a straightforward search, we are interested in finding a path from the start state to the goal state. In planning, however, we usually add many constraints to the path being searched. We might, for example, be interested in the least-cost path where operators have costs associated with them. This is precisely the case in the *crates problem* [34, p.47], where a warehouse is nearly full of heavy crates which must be pushed in order to move one from the back to the front door. The goal here is not to find some way of moving the crate, but to find the "path of least effort". While the planning literature abounds with algorithms that find constrained paths to the goal state, we feel that there is little emphasis on the path itself. In the most general case, what we are searching for is a constrained path with the emphasis on the path rather than the goal state. We can envision, for example, that the goal of our search is to find a path which does not terminate with some fixed "goal" state, but rather ends with a directed loop through the search space. We have addressed this possibility in [12] and give an account of our approach in Chapter 3.

To close our discussion of planning/searching, we include another quote from the same Nilsson article [54] from the point of view of the "core" system and specifically its Predicting and Planning capability:

> *The core system needs the ability to make predictions of future perceived states that will result from proposed actions taken in present states. These predictions form the basis for making plans of actions to achieve goals.*

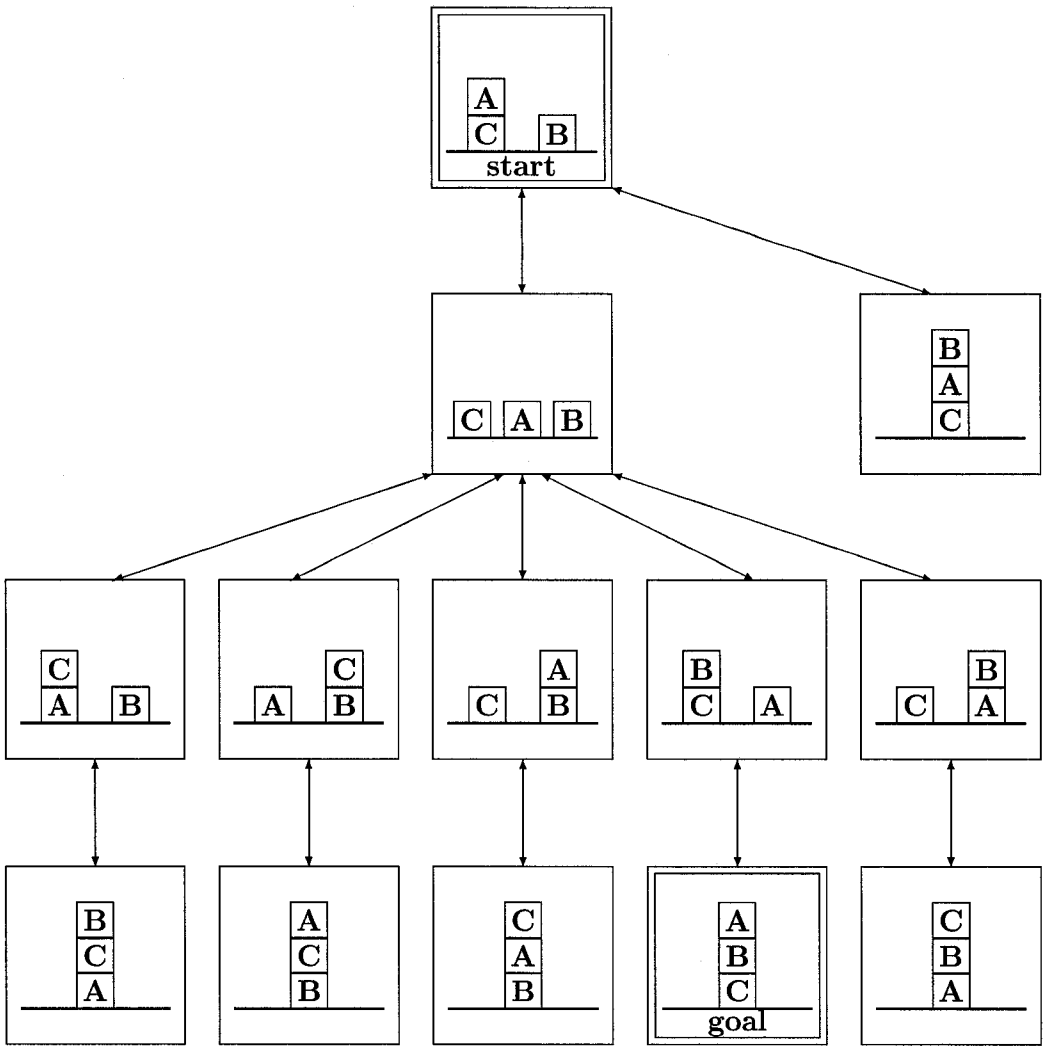

Figure 2.3: Search graph in the BLOCKS world

It is not difficult to recognize that the above quote is using the terminology of Reinforcement Learning, which allows us to smoothly segue into the next section.

2.4 Mixed Approaches: Putting Learning and Planning Together

2.4.1 Reinforcement Learning

In his seminal paper [89], Turing anticipated the need for what we now call Reinforcement Learning [81]:

> *We normally associate punishments and rewards with the teaching process. Some simple child machines can be constructed or programmed on this sort of principle. The machine has to be so constructed that events which shortly preceded the occurrence of a punishment signal are unlikely to be repeated, whereas a reward signal increased the probability of repetition of the events which led up to it. ... The use of punishments and rewards can at best be a part of the teaching process. Roughly speaking, if the teacher has no other means of communicating to the pupil, the amount of information which can reach him does not exceed the total number of rewards and punishments applied.*

The classical Reinforcement Learning (RL) [81] scheme formalizes an interaction process between a single agent and its environment. The agent observes the current state of the environment s from a set of states S (typically finite) and selects one action a from the set A (also, typically finite) of available actions. The action is, in turn, communicated to the environment, which responds by changing its state and issuing a reward or penalty in the form of a real-valued reinforcement signal. If the signal value is positive (*reward*), the agent should take it as an indication that it is doing well, and conversely, if the signal is

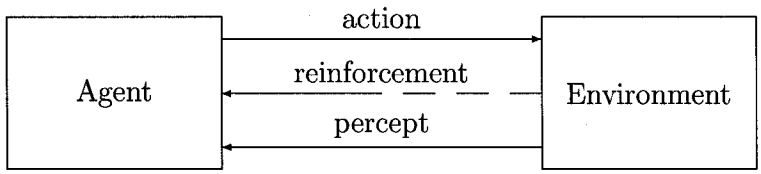

Figure 2.4: Classical RL scheme

negative (*penalty*) the agent should infer that it is doing poorly. The relative value of the reinforcement will determine just how well or poorly the agent performs. A sensible agent will likely learn from the reinforcement signal and adjust its behaviour (*policy*) accordingly. This cycle of sense-act-learn repeats itself until the interaction is terminated by either a learning failure, or a natural completion of the training episode (e.g. a game is finished). Figure 2.4 depicts the classical RL feedback loop.

The classical RL framework is intended to simplify the development of many algorithms that can be compared in terms of their performance on a variety of tasks by choosing the appropriate (and usually simulated) environment. The above description of the interaction omits a number of details crucial to a successful implementation. Many of the general RL algorithms, for example, assume that the state of the environment is expressed by a natural number in a known range (i.e. the total number of states is known in advance). This may present a difficulty if we don't know, *a priori*, how many different possible states the environment can be in (e.g. when the environment is a chess game). Furthermore, even if we know the total number of states, it is often quite awkward and inefficient to represent them as a single number. In a game of tic-tac-toe, for example, we can encode every game state into a single number. By doing so, however, we eliminate many spacial symmetries inherent in the game, making learning much less efficient. In other cases, agent actions may be better represented by a number of independent "subactions" instead of a single action a. A robot, for example, may roll in a certain direction and emit a sound at the same time. We

will address these issues in Chapter 3.

The RL paradigm is usually categorized as a representative of a mixture of supervised and unsupervised learning approaches. In this model, the environment is considered to be a teacher of sorts. Unlike supervised learning, the task (e.g. to classify a new instance) is not known to the learning system *a priori*, and must itself be induced from the received reinforcements. Thus, in RL, learning must occur at several levels simultaneously. First of all, more compact descriptions of the environment states can be produced by learning which combinations of features in the original descriptions are relevant to successful prediction of the environment's dynamics. Secondly, the rules that govern the environment's dynamics must be acquired, and temporal generalization has to be accomplished. Finally, as both of these inductive processes take place, the agent must learn what the task itself is, by generalizing from the reinforcements, while taking into account the environment's context.

It can also be said that RL uses a form of unsupervised learning. While the reinforcement signal classifies the agent's behaviours as "good" or "bad" in a supervised fashion, any successful agent, that constructs a model of the environment, must classify the latter's responses according to *its* own understanding of what is relevant to the induced task.

The RL framework stands out from others presented so far in this Chapter, because it[7] accommodates building complete and general systems capable of exhibiting "rational" or "intelligent" behaviours when dealing with stochastic environments. RL algorithms not only learn, but also plan[8] their actions based on acquired knowledge and guided by their learned goal.

---

[7]The framework of Learning Automata, which is discussed below, possesses similar characteristics with a slightly different emphasis.

[8]As a key ingredient to intelligence, planning is discussed in Section 2.3.

2.4.1.1 RL Applications

Due to the generality of the RL framework, it can accommodate a wide variety of applications. Among the earliest successes of RL-like methods were the MENACE[9] tic-tac-toe player by Michie [46], and the checkers program by Samuel [74]. At the time of their creation, RL was not perceived as a distinct direction of AI research[10] separate from other AI approaches. Yet many of these systems' features can now be recognized as belonging to the RL domain. Michie's use of coloured beads to represent the action-selection probabilities is virtually the same as the state-action values used today in Q-learning (described in Section 2.4.1.6). Samuel's use of backup procedures for value (heuristic, in his case) functions combined by a form of learning is reminiscent of *temporal-difference* learning methods, popular in RL today. The checkers program created by Samuel can, in fact, be considered to be the precursor of the successful TD-Gammon backgammon playing program due to Tesauro [83]. TD-Gammon uses a variant of temporal-difference learning in combination with a backpropagation neural network that approximates the value function explained below.

So far, the mentioned applications all belong to the game playing domain. Soon after developing MENACE, Michie demonstrated a successful application of RL techniques to an adaptive control problem of balancing a pole on a cart. The algorithm, called BOXES [47], learned to balance a pole mounted with a single degree of freedom on a cart moving on rail tracks (Figure 2.5). The task was to keep the pole in the upright position for the longest possible time. The experiments showed that after approximately 70 hours of training, BOXES was able to balance the pole for an impressive period of 27 minutes!

Another example of a similar robotic control problem is the one faced by the *Acrobot*, a two-link robot resembling a gymnast swinging on a high bar (see Figure 2.6). The torque is applied only to the joint between the two links. Sutton [81, pp. 270–274] successfully

[9]Stands for Matchbox Educable Naughts and Crosses Engine.

[10]The term *reinforcement learning* was already in use, however.

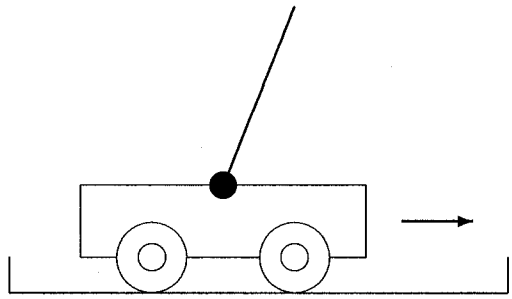

Figure 2.5: Pole and cart system. In this configuration, to keep the pole balanced, the cart must move right.

applied model-free on-line RL methods in a simulated environment.

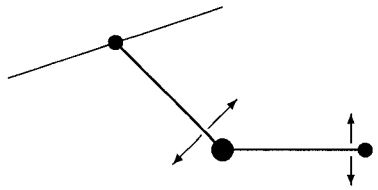

Figure 2.6: *Acrobot.* The goal is to swing the "feet" a certain height above the high bar.

While the above problems faced by BOXES and the *Acrobot* can be solved using non-RL methods, consider now a different problem of elevator dispatching. In a large building with multiple elevators, at any given time, many passengers are waiting on different floors. The problem considered here is to decide which elevators should pick up which passengers, and in what sequence these operations are to be done in order to minimize the waiting time. This is a good example of a stochastic optimal control problem of economic importance. This problem is intractable by classical techniques because a suitable model for the distribution of dispatch requests is not easily obtained. Additionally, even if we had such a model, it may become inappropriate if certain conditions within the building change, e.g. a large company clearing its office space in the building. In this instance, the number of passengers

on some floors would be affected. Crites and Barto [24] applied RL techniques to solve this problem by using a variant of Q-learning. According to their experiments RL techniques outperformed many known heuristic dispatch methods and have compared favourably with other non-proprietary techniques used in their simulations.

2.4.1.2 Markovian Assumption

A variety of RL algorithms are in existence — a survey of which can be found in [27]. Most are intended for discrete-time interactions, and many make an assumption, which is violated in most practical applications, that the environment is fully observable. A related and crucial assumption that is almost universally made, is that the environment dynamics satisfy the so-called *Markov* property. In the following discussion, if the dynamics of the environment satisfy the Markov property, we call such an environment — a Markov environment.

In short, the evolution of a Markov environment depends only on its current state and is independent of the history of its past interactions. Such a property is often rephrased as the "independence of path" property, because all the agent needs to know to predict the future behaviour of the environment is the current state signal. In other words, the agent does not need to take into account how it arrived at this state — it can completely ignore the exact path that led to the current state in making a decision about which course of action is to be chosen. This is not to say that the actions taken in the past do not matter at all. In fact, the only way the agent can learn is to learn on the basis of its past interactions with the environment. The Markov property merely guarantees that the actions taken in the current state of the environment can be independent of the path that led to this current state. An agent interacting with a Markov environment is said to face a problem which is often called a Markov Decision Process (MDP) (introduced by Bellman [16]). If the number of states and actions are finite, the MDP is also called finite.

We present below a more formal definition of the Markov property as it pertains to

Reinforcement Learning. In the interest of simplifying the notation, we are going to assume that we are dealing with a finite number of states, rewards, and actions. Let us consider the state the environment is going to switch to at time $t+1$ in response to the action taken at time t. In the most general case, this response may depend on the entire history of what transpired before time t, forcing us to consider the complete probability distribution:

$$P\{s_{t+1} = s', r_{t+1} = r | s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0\}, \tag{2.1}$$

for all s', r, and all possible values of the past events: s_t, a_t, r_t, $\ldots$, r_1, s_0, a_0. If we are dealing with a Markov environment, however, we can ignore past events and simplify the probability distribution to:

$$P\{s_{t+1} = s', r_{t+1} = r | s_t, a_t\}, \tag{2.2}$$

for all s', r, s_t, and a_t. We would like to bring the reader's attention to the fact that in RL (as exemplified by Figure 2.4) the reinforcement signal is assumed to be part of the environment, which is a reasonable assumption to make at the chosen level of granularity where the environment essentially encompasses everything outside of the agent's control. Under such an assumption, the Markov property of the environment also covers the dynamics of the reinforcement signal.

Equating probabilities (2.1) and (2.2) for all s', r, and s_t, a_t, r_t, $\ldots$, r_1, s_0, a_0 is a necessary and sufficient condition for the environment to possess the Markov property. By successively applying this equation to future states, one can effectively predict the entire evolution of the environment, including expected rewards. Specifically, such a prediction can be made only on the basis of observing the current state, irrespective of the entire history that led to this state. Moreover, the knowledge of such a history cannot make the prediction more accurate!

The assumption that the environment is Markov is often crucial to successful analysis of RL algorithms, especially the convergence of these algorithms to optimal policies[11]. Except

[11]For the proof of the convergence of Q-learning we refer the reader to Section 2.4.1.6.

for simple environments that are carefully chosen to test the algorithms on, this assumption unfortunately, does not hold in practice. The primary reason for this is the fact that for many practical problems the environment cannot be fully observable, i.e. many relevant aspects of the environment remain hidden (unobserved) by the agent. If it were possible to observe the complete state of the environment, the Markov property would have likely been satisfied, as most laws of physics (describing the dynamics of real environments) are in fact based on Markov assumptions. Even in situations where the environment appears to be fully observable for the purposes of predicting state transitions, it may not be really fully observable since the rewards do not satisfy the Markov requirement. For example, the goal assigned to the agent may require the agent to retrace a particular trajectory through the state space. In this case, the reward will not only depend on the current state but also on the trajectory that led to that state.

Since the Markov environment assumption is used in convergence proofs of some algorithms (see Section 2.4.1.6), these algorithms are no longer guaranteed to converge in situations where the Markov property is not satisfied. Experiments confirm, however, that even under such circumstances, the algorithms often continue to exhibit convergence, especially when the environment is "nearly" Markovian. A thorough understanding of the theory, when the Markov assumption holds, lays an essential foundation for extending it to more realistic non-Markov environments. The theory of MDPs is treated in [18] and [70], to name a few.

2.4.1.3 Value Function

An RL algorithm attempts to learn an action selection policy π, which is a mapping from the states of the environment to the actions chosen by the algorithm. More generally, a policy maps to a probability distribution over the set of possible actions in a given state, i.e. the probabilities of taking each action in a given state.

To successfully navigate the environment, an RL algorithm usually needs some way of distinguishing between "good" and "bad" states, which are, respectively, states with high and low potential for future rewards. The sum of all future rewards that the agent can expect is known in the literature as the expected *return* [81, p.57], which is a quantity that the agent wants to maximize.

To define it more formally, we need to consider the nature of the agent's interaction with the environment. In many applications there is a natural notion of a final time step which concludes the interaction between the agent and the environment. For example, all multi-player games ordinarily end with a win of one side over the opponents or possibly with a draw[12]. The existence of a final time step implies that the interaction ends within a finite number of time steps. Learning tasks of this nature are called *episodic*. The state of the environment reached during the final time step of an episodic task is called the *terminal* state [81, p.58]. For episodic tasks we can define the return as:

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \cdots + r_T, \tag{2.3}$$

where T is the final time step.

There exist, however, many tasks that are non-episodic in nature, where the agent-environment interaction does not naturally break into identifiable episodes. A typical example of such a task is the pole-balancing problem mentioned earlier. While the individual learning trials may and, indeed, often do end in a failure (an agent is still learning), the ultimate goal is to continue interacting with the environment indefinitely. Learning tasks with such objectives are called *continuous*[13]. The simple definition of return shown above presents a difficulty with respect to continuous tasks. In the absence of the final time step T, the simple summation of all the rewards can potentially become infinite. The task of

[12]Even stalemates in chess can be recognized as such and allow the players to terminate a game.

[13]The MDP literature often uses the terminology of *finite-horizon*, *indefinite-horizon* and *infinite-horizon* tasks. It is the latter two cases that correspond to episodic and continuous tasks in [81].

maximizing an already infinite return is, therefore, poorly defined, especially for environments where every possible policy leads to infinite returns. Several approaches are possible for continuous tasks. Instead of maximizing the absolute sum of rewards, we may, for example, want to maximize the average reward per unit of time, i.e. the instantaneous expected reward. Another popular method is to discount rewards that are too far in the future in favor of more immediate rewards. Thus, the return can be defined as:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots \equiv \sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i}, \tag{2.4}$$

where $0 \leq \gamma \leq 1$ is the discount rate parameter, determining the value of future rewards relative to the immediate rewards. In other words, a reward received in the next time step will be perceived as $1/\gamma^{j-1}$ times more valuable then the same reward obtained j time steps into the future. Such a definition of a return has a nice mathematical property of leading to a convergent sum when $\gamma < 1$, when the reward sequence $\{r_i\}$ is bounded. If we set γ to 0, the agent will completely ignore all future rewards and will try to converge to a completely "nearsighted" or greedy policy of maximizing the immediate reward from every encountered state. On the other end of the spectrum, if γ is set to 1, the definition degenerates to Equation (2.3) where the agent does not discriminate between immediate and future rewards.

Consider Equation (2.4) as our working definition of return. The expected return depends on the actions that the agent will select, i.e. the current policy. We can, thus, arrive at a notion of a *value* of a state as an expected return when following a specific policy π:

$$V^{\pi}(s) = E_{\pi}\{R_t | s_t = s\} \equiv E_{\pi}\left\{ \sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i} \middle| s_t = s \right\}, \tag{2.5}$$

where E_{π} is the expected value given that the agent follows policy π from state s.

If the dynamics of the environment are unknown, the value function can be estimated from experience [81, p.69]. Simple methods can be employed (see Section 2.4.1.5 for more details) to maintain an average of actual returns for each encountered state. Popular examples of

more sophisticated methods of recording and maintaining the experience of an agent include Q-learning (Section 2.4.1.6) and Temporal-Difference Learning (Section 2.4.1.7), in general.

If, on the other hand, the dynamics of the environment are already known, the value function can be computed using Dynamic Programming approaches as discussed in Section 2.4.1.4.

The ability to compute/estimate the value function does not, by itself, solve the RL problem. An RL algorithm must find a policy that offers the greatest rewards, i.e. an *optimal* policy. As it turns out [81, p.75], the notion of an optimal policy can be formalized with ease for finite MDPs. In particular, it can be shown that there exists a partial ordering over all possible policies. Specifically,

$$\pi \succeq \pi' \quad \equiv \quad \forall s \in S, \quad V^{\pi}(s) \geq V^{\pi'}(s).$$

In other words, policy π is better than or equal to policy π' if and only if the value of each state under π is greater than or equal to the value of the same state. The optimal policy π^* is better than or equal to all other policies, i.e. :

$$\forall \pi, \quad \pi^* \succeq \pi.$$

There may exist several different equally optimal policies, but all of them would have exactly the same value function, denoted V^*, and defined as:

$$V^*(s) \equiv max_{\pi} V^{\pi}(s), \quad \forall s \in S.$$

While an optimal policy is clearly an ideal solution to the RL problem, it is often outside of what is practical. This is due to the limitations of the algorithms used (e.g. Markov assumption) or the prohibitive computational cost. Possible approaches in such situations are discussed in Section 2.4.1.8.

2.4.1.4 Policy Iteration / Dynamic Programming

Let us assume that we are dealing with a very well-known environment, in that its dynamics are fully specified. By dynamics we mean both the set of transition probabilities $\mathcal{P}^a_{ss'}$ and the set of expected immediate rewards $\mathcal{R}^a_{ss'}$, formally defined as:

$$\mathcal{P}^a_{ss'} = \Pr\{s_{t+1} = s' | s_t = s, a_t = a\} \tag{2.6}$$

$$\mathcal{R}^a_{ss'} = E\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\}, \tag{2.7}$$

for all $s \in \mathcal{S}$, $a \in \mathcal{A}$ and $s' \in \mathcal{S}^+$, where $\mathcal{S}^+$ consists of $\mathcal{S}$ and all the terminal states if the task is episodic.

If we additionally make the Markov assumption, both $\mathcal{P}^a_{ss'}$ and $\mathcal{R}^a_{ss'}$ give us complete information necessary to forecast the evolution of the environment from any starting state, given a particular fixed policy that the agent is following.

In particular, we can attempt to compute the value function given the above knowledge of the environment's dynamics and an arbitrary and fixed policy π. The best understood approach to such a computation is based on using a form of a Dynamic Programming algorithm. The term "Dynamic Programming" (henceforth abbreviated as DP) was coined by Bellman who demonstrated [15] the wide applicability of this class of algorithms. The idea of potentially applying DP to AI is attributed to Minsky [48], while the connection between DP and RL first appeared in works by Andreae and Werbos (see, for example, [9], [92]). Watkins was the first to explicitly connect DP and RL by characterizing some RL methods as "incremental" DP [91].

Given the environment dynamics of Equation (2.6) and the definition of the value function under policy π, Equation (2.5), we can rewrite the expectation as a summation (assuming finite $\mathcal{S}$ and $\mathcal{A}$):

$$V^{\pi}(s) = E_{\pi}\left\{\sum_{i=1}^{\infty}\gamma^{i-1}r_{t+i}\middle|\, s_t = s\right\} \tag{2.8}$$

$$= E_{\pi}\left\{r_{t+1} + \gamma V^{\pi}(s_{t+1})\middle|\, s_t = s\right\} \tag{2.9}$$

$$= \sum_{a}\pi(s,a)\sum_{s'}\mathcal{P}^{a}_{ss'}\left[\mathcal{R}^{a}_{ss'} + \gamma V^{\pi}(s')\right], \tag{2.10}$$

where $\pi(s,a)$ is the probability of taking an action a in a state s under policy π. The above equation (2.8) is known as the *Bellman* equation. For V^{π} to exist and be unique, it is sufficient for either $\gamma < 1$, or for the task to be episodic (i.e. terminating) from all states under this policy [81, p.90].

The above equation, in fact, describes a set of $|\mathcal{S}|$ linear equations with $|\mathcal{S}|$ unknowns. Any of the number of ways of solving this set of linear equations would produce the value function for a given policy. This step is known in the literature as the *Policy Evaluation* or the *Prediction Problem.* Since $|\mathcal{S}|$ is usually quite large for problems of any interesting size, iterative solution methods are often employed. If the Bellman equation (2.8) were rewritten as an update rule:

$$V_{k+1}(s) \Leftarrow \sum_{a}\pi(s,a)\sum_{s'}\mathcal{P}^{a}_{ss'}\left[\mathcal{R}^{a}_{ss'} + \gamma V^{\pi}_{k}(s')\right], \tag{2.11}$$

then V^{π} is a fixed point, i.e. $V^{\pi}_{k} \rightarrow V^{\pi}$ as $k \rightarrow \infty$. An algorithm based on this update rule is known as an *iterative policy evaluation.*

The ultimate goal of an RL algorithm, however, is to find a better policy, and not just that of being able to evaluate a particular policy. We need to be able to compare different policies and be able to choose in favour of the one that induces a more favourable value function. For the sake of simplicity, let us consider deterministic policies only. Instead of imposing a probability distribution over all actions that can be chosen, such policies always choose one particular action for a given state, i.e. all the probability is concentrated in one state.

Once we compute the value of all the states under the current policy, π, we can discover obvious inefficiencies in the policy. For example, let us assume that from state s_i, actions a_1 and a_2 can be taken, leading deterministically and correspondingly to states s_j and s_k. Suppose that $\pi(s) = a_1$, but $V^\pi(s_j) < V^\pi(s_k)$. It would seem perfectly reasonable to then switch to a new policy π', such that

$$\pi'(s) = \begin{cases} a_2 & : \quad s = s_i \\ \pi(s) & : \quad s \neq s_i. \end{cases}$$

This change in policy is known as a *policy improvement*. Due to the so-called *policy improvement theorem*, it can be shown that the resulting $V^{\pi'}(s) > V^\pi(s)$.

In the simple example above, we considered only a single greedy improvement at some state s_i. Interestingly, we can make the same greedy improvements at every state s. The surprising outcome of such greedy improvements is that if a policy cannot be improved any further, i.e. $V^{\pi'}(s) = V^\pi(s)$, it can also be shown [81, p.96] that such a policy must be optimal as is the corresponding value function, i.e.

$$V^{\pi'} = V^* \quad \text{and} \quad \pi' = \pi^*.$$

This leads us to a *policy iteration* algorithm where we start out with some initial policy π_0, evaluate it, improve it, and arrive at a better policy π_1, which we, in turn, evaluate and improve. A sequence of monotonically improving policies and value functions can thus be obtained:

$$\pi_0 \xrightarrow{E} V^{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} V^{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \cdots \xrightarrow{I} \pi^* \xrightarrow{E} V^*,$$

where $\xrightarrow{E}$ denotes the policy evaluation step and $\xrightarrow{I}$ denotes the policy improvement step. Since a finite MDP has a finite number of deterministic policies, this algorithm converges to an optimal policy in a finite number of steps [81, p.97].

2.4.1.5 Monte-Carlo Methods

Given the current state of the environment, the agent is to select an action that will bring it the largest cumulative reward. If the environment dynamics are known, the task is reduced to finding an optimal policy. This can be solved by DP methods pioneered by Bellman [15]. The DP algorithms of Policy Iteration and Value Iteration essentially perform a search in the space of policies, in order to converge to an optimal one. No on-line interaction between an agent and its environment is needed to compute the optimal policy. These algorithms suffer from the so-called *curse of dimensionality* because they require a sweep over the entire state space for each iteration, and the state space grows exponentially with the number of state variables. Despite this handicap, DP methods can be considered reasonably efficient, as their worst-case analysis yields polynomial time performance in the number of states and actions. These methods are also attractive, as they are the ones most studied, from a formal mathematical point of view.

Even in cases where we potentially should have full knowledge of the environment, it may be difficult to obtain an explicit model necessary for the DP methods. As mentioned in [81, p.114], we know all the rules that govern the game of blackjack, yet computing the explicit $\mathcal{P}^a_{ss'}$ and $\mathcal{R}^a_{ss'}$ is difficult and error-prone. For example, what is the expected reward if the player's hand contains a king and a four (for a total sum of 14) when that player decides to "stay", while the dealer's single showing card is an eight? Before a DP method can be applied, all such expected rewards must be fully computed.

In general, however, we do not have the model of the environment available and, consequently, the model of the reinforcements is also not known. Thus, we are forced to learn from experience, and base our decisions on observed sample sequences of states, actions, and rewards. These constraints naturally lead us to the so-called *Monte Carlo* (abbreviated as MC) methods based on averaging sample returns. Since we need to ensure that interaction with an environment will eventually yield sample returns, MC methods are well-defined for

episodic tasks only. In such tasks, the interactions with the environment eventually terminate regardless of the choice of actions.

The term "Monte Carlo" refers to the famous Monte Carlo casino, and was reportedly first used in a scientific setting when physicists at Los Alamos studied games of chance to help them understand complex physical phenomena. Michie and Chambers used Monte Carlo methods for BOXES [47] (See Section 2.4.1.1), while Narendra and Wheeler [52] studied a Monte Carlo method for ergodic finite Markov chains in the context of learning automata (See Section 2.4.2 for a review of the Learning Automata field). Singh and Sutton [78] proved results relating MC methods to reinforcement learning.

Let us initially consider the problem of estimating the value function given a particular policy. Since the value of a state is precisely the expected return starting from this state, i.e. an expected cumulative future discounted reward, a natural way to estimate these returns from experience is to simply average the returns observed after repeated visits to a state.

If the model of the environment is not known, however, our estimates of the value function are not sufficient to formulate a policy. If we had a model, we could "expand" the current state into a set of "following" states and obtain value estimates for each of these "next" states. Then we could choose the action that leads us to the best combination of "next" state's value and an immediate return. In the absence of the model, we can choose to estimate *action* value functions rather than the state value functions. Instead of giving an overall value for a particular state, an action value function yields the value of a combination of a state and an action chosen from this state. The action function is typically denoted as $Q(s, a)$. The following gives the relationship between the state value function, V, and the action value function Q:

$$V^{\pi}(s) = \max_{a} Q^{\pi}(s, a). \tag{2.12}$$

Every time an agent "visits" a state-action pair in an on-line interactive fashion, it records

the return it receives by following the current policy until the end of the episode. Over time these returns are averaged, and we can obtain the value of each state-action pair visited using the current policy. This value for state s and action a can, thus, be formally defined as:

$$Q^{\pi}(s,a) \equiv r(s,a) + \gamma V^{\pi}(s'), \tag{2.13}$$

where $V^{*}(s)$ is the value of state s under the optimal policy, s' is the next state and $r(s,a)$ is the immediate reward after action a is performed in s.

In MC methods, the Q estimates are updated at the end of each episode, because overall returns must be averaged. It has been observed, however, that one can update the Q estimates after every single interaction step. Learning the Q function in this fashion forms the basis for the well-known Q-learning algorithm [91] described in Section 2.4.1.6 below.

Once we estimate the action value function for a given policy π, we can obtain a better policy by choosing a greedy improvement:

$$\pi'(s) = \arg\max_{a} Q(s,a). \tag{2.14}$$

2.4.1.6 Q-learning

At the time when it was first proposed [91] the Q-learning algorithm was a major breakthrough in RL research. In Q-learning, the current estimate of the Q function is represented as an $S \times A$ table, where S is the total number of environment states, and A is the total number of actions that an agent can perform. By combining Equations (2.12) and (2.13),

$$Q^{\pi}(s,a) \equiv r(s,a) + \gamma \max_{\bar{a}} Q^{\pi}(s',\bar{a}), \tag{2.15}$$

we obtain an equation that forms the basis for the update in Q-learning. Every time the agent receives a reinforcement, it updates the appropriate entries in the table, and then selects the best action based on the current Q values. The outline for the algorithm is given in Figure 2.7.

For each s, a

Initialize the table entry $\hat{Q}(s, a)$ to zero.

Repeat forever:

Observe the current state s

Select an action a and execute it

Receive the immediate reward r

Observe the new state s'

Update the table entry for $\hat{Q}(s, a)$ as follows:

$\hat{Q}(s, a) \leftarrow r + \gamma \max_{\bar{a}} \hat{Q}(s', \bar{a})$

End Repeat

Figure 2.7: Basic Q-learning algorithm. At the end of each episode the environment is assumed to automatically reset the initial state.

The update rule of the Q table is very similar to Equation (2.15):

$$\hat{Q}(s, a) \leftarrow r + \gamma \max_{\bar{a}} \hat{Q}(s', \bar{a}), \tag{2.16}$$

where $\hat{Q}(s, a)$ is the current estimate of the true $Q(s, a)$ of Equation (2.13). Essentially $\max_{\bar{a}} \hat{Q}(s', \bar{a})$ is taken as the current estimate of $V^*(s')$, or the estimated value of the state if the best action is chosen from it. If we are dealing with an episodic task and the state s' is a terminal one, then $\hat{Q}(s', \bar{a})$ is defined as zero.

Q-learning is known as the off-policy algorithm because the updating rule moves the Q value towards $\max_{\bar{a}} \hat{Q}(s', \bar{a})$ rather than the $\hat{Q}(s', a')$. Here $a' = \pi(s')$, i.e. the action chosen in the next state s' is based on the current policy π, leading to the off-policy / on-policy distinction. The on-policy version of the update rule is:

$$\hat{Q}(s, a) \leftarrow r + \gamma \hat{Q}(s', a'). \tag{2.17}$$

This rule forms the basis of the related Sarsa[14] algorithm [81, p.145]. The reason Q-learning is so important in the RL field is that, unlike Sarsa, it directly approximates Q^* independent of the current policy. This simplifies the analysis of the algorithm which, in turn, allows us to prove convergence. We include the proof below.

Note that the step where the action is selected does not specify *how* the action is selected. For instance, we could select the action in a totally random fashion. Clearly, in this case the behaviour exhibited by an agent would also be random. Despite such a discouraging behaviour, the algorithm will still converge to the optimal policy through the backups to the Q values. Consequently, after a long learning phase during which the behaviour is random, we can switch to a completely greedy policy of always choosing the action with the largest Q value[15]. Usually we want to blend these two extreme approaches (see below for description of ϵ-greedy and soft-max approaches). Whatever approach we choose, for this algorithm to converge to an optimal policy, we need to ensure that each state-action pair is visited infinitely often. Let us now address the convergence of Q-learning in a more formal setting. We include the formal proof of convergence which is essentially reproduced from [50, pp.377 – 379]:

Theorem 2.1 ***Convergence of Q-learning for deterministic MDPs.*** *Consider a Q-learning agent in a deterministic MDP with bounded rewards, that is* $(\forall s, a)|r(s,a)| \leq c$. *The Q-learning agent initializes its Q table to arbitrary finite values and uses the backup rule of equation (2.13), where* $0 \leq \gamma < 1$. *Let* $\hat{Q}_n(s,a)$ *denote the agent's estimate of* $\hat{Q}(s,a)$ *following the nth update. If each state-action pair is visited infinitely often, then* $\hat{Q}_n(s,a) \rightarrow Q(s,a)$ *as* $n \rightarrow \infty$, *for all s and a.*

Proof. Since each state-action transition occurs infinitely often, consider consecutive intervals during which each state-action transition occurs at least once. The proof consists

[14]Called so for the sequence of symbols in the update rule: s, a, r, s', a'.

[15]If several actions have the same exact Q value, we can still choose randomly among them.

of showing that the maximum error over all entries in the $\hat{Q}$ table is reduced by at least a factor of γ during each such interval. $\hat{Q}_n$ is the agent's table of estimated Q values after n updates. Let Δ_n be the maximum error in $\hat{Q}$ implying that:

$$\Delta_n \equiv \max_{s,a} |\hat{Q}_n(s,a) - Q(s,a)| \tag{2.18}$$

Below we use s' to denote $\delta(s,a)$ or the state resulting from choosing action a in state s. Now for any table entry $\hat{Q}_n(s,a)$ that is updated on iteration $n+1$, the magnitude of the error in the revised estimate $\hat{Q}_{n+1}(s,a)$ is:

$$\begin{aligned}
|\hat{Q}_{n+1}(s,a) - Q(s,a)| &= |(r + \gamma \max_{a'} \hat{Q}_n(s',a')) - (r + \gamma \max_{a'} Q(s',a'))| \\
&= \gamma |\max_{a'} \hat{Q}_n(s',a') - \max_{a'} Q(s',a')| \\
&\leq \gamma \max_{a'} |\hat{Q}_n(s',a') - Q(s',a')| \\
&\leq \gamma \max_{s'',a'} |\hat{Q}_n(s'',a') - Q(s'',a')| \\
&\leq \gamma \Delta_n.
\end{aligned}$$

In the above derivation, the third line follows from the second because for any two functions, f_1 and f_2, the following inequality holds:

$$|\max_a f_1(a) - \max_a f_2(a)| \leq \max_a |f_1(a) - f_2(a)|. \tag{2.19}$$

In going from the third line to the fourth line above, note that we introduce a new variable s'' over which the maximization is performed. This is legitimate because the maximum value will be at least as great when we allow this additional variable to vary. Note that by introducing this variable we obtain an expression that matches the definition of Δ_n.

Thus, the updated $\hat{Q}_{n+1}(s,a)$ for any s, a is at most γ times the maximum error in the $\hat{Q}_n$ table, Δ_n. The largest error in the initial table, Δ_0, is bounded because values of $\hat{Q}_0(s,a)$ and $Q(s,a)$ are bounded for all s and a. Now after the first interval during which each s, a is visited, the largest error in the table will be at most $\gamma\Delta_0$. After k such intervals, the error

will be at most $\gamma^k \Delta_0$. The number of such intervals is infinite because each state is visited infinitely often. Since $\gamma < 1$, $\Delta_n \rightarrow 0$ as $n \rightarrow \infty$.

□

The above proof assumes that the environment is deterministic. This assumption, however, is violated in many practical applications[16]. A non-deterministic update rule must, therefore, be used to assure convergence, and in this case, the update rule is modified [50, p.382] to make a more gradual change to the Q estimate as shown below:

$$\hat{Q}_n(s,a) \leftarrow (1-\alpha_n)\hat{Q}_{n-1}(s,a) + \alpha_n[r + \max_{a'} \hat{Q}_{n-1}(s',a')],$$

where $0 \leq \alpha_n \leq 1$ is the learning rate, and $\hat{Q}_n(s,a)$ is the agent's estimate of Q on the n^{th} iteration of the algorithm. The learning rate itself can be gradually reduced for states that have been visited many times before, according to the following formula:

$$\alpha_n = \frac{1}{1 + visits_n(s,a)},$$

where $visits_n(s,a)$ is the total number of times this state-action pair has been visited up to and including the n^{th} iteration. This evolution of the learning rate ensures that the algorithm makes more conservative changes to the Q estimate for state-action pairs that have been encountered many times before, and which are, thus, presumably close to the true value of Q.

As already mentioned, the Q-learning algorithm described above omits an important detail of how the actions are selected. The RL literature describes several ways of balancing *exploration* and *exploitation*, including greedy, ϵ-greedy and soft max [81, pp.28 – 31] methods. In the more flexible *soft max* approach, the actions are chosen according to a probability distribution, which depends on the current estimates of Q values and a special parameter τ,

[16]See Section 4.4, for an example in the tic-tac-toe domain.

called the *temperature*. The recommended soft max distribution [81, pp.28 – 31] is computed as:

$$P(a_i|s) = \frac{e^{Q(s,a_i)/\tau}}{\sum_j e^{Q(s,a_j)/\tau}}.$$

The reader will observe the similarity between the temperature schedule and the techniques used in *simulated annealing*. When the temperature is close to 0, the agent will follow a greedy policy of always choosing the most promising action. As $\tau \rightarrow \infty$, the actions are selected at random according to a uniform distribution.

If we take a close look at the Q-learning algorithm, we will discover that it is possible to change the order of updating the Q estimates without affecting the overall convergence of the algorithm. While the learning agent is interacting with the environment, it can simply record its actions, and the resulting state transitions and rewards for the duration of the episode (or for a fixed number of steps) without performing any updates. Once the episode (or the predetermined number of steps) is over, the updates can be applied in the reverse order, starting from the last state transition as follows:

$$\begin{array}{lll}
1. & Q(s_{n-1}, a_{n-1}) & \leftarrow r_{n-1} + \gamma \max_a Q(s_n, a) \\
2. & Q(s_{n-2}, a_{n-2}) & \leftarrow r_{n-2} + \gamma \max_a Q(s_{n-1}, a) \\
\vdots & \vdots & \vdots \\
n. & Q(s_0, a_0) & \leftarrow r_0 + \gamma \max_a Q(s_1, a).
\end{array}$$

This updating scheme is called *batch* updating. The advantage of batch updating is that after the episode, the *return* (a possibly discounted sum of all the rewards) will be propagated all the way to the starting state/action pair s_0 and a_0, whereas if updates are performed immediately after each state transition, several episodes would have been required to achieve the same effect. The clear disadvantage of batch updating is the extra memory required to store everything that transpires in an episode. This technique will be clearly inappropriate for very long episodes, because, not only would it require large amounts of additional memory,

but it would also permit no training during these long episodes.

2.4.1.7 Temporal Difference Learning

As it turns out, Q-learning is a special case of a more general class of algorithms called *temporal-difference* (TD) learning [81, ch.6], which combines ideas used in Monte Carlo (learning from experience without having a model of environment) and DP methods (learning a guess from a guess — bootstrapping). Let us denote s_t, a_t, and r_t as the state, the action, and the resulting reinforcement that had happened during time step t. In updating the "current" Q-estimate $\hat{Q}(s_t, a_t)$, Q-learning uses the estimates of Q values for the following state s_{t+1}, and the single reinforcement r_t received at time t. This is known as a one-step update. The TD(λ) algorithm generalizes the updating rule to include the Q estimates for the states following s_{t+1} as well as the reinforcements received at time $t+1$ and later.

The following table lists the possible multi-step updates:

$$n = 1 \quad Q(s_t, a_t) \leftarrow r_t + \gamma \max_a Q(s_{t+n}, a) \tag{2.20}$$

$$n = 2 \quad Q(s_t, a_t) \leftarrow r_t + \gamma r_{t+1} + \gamma^n \max_a Q(s_{t+n}, a) \tag{2.21}$$

$$\vdots \tag{2.22}$$

$$n = N \quad Q(s_t, a_t) \leftarrow r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^n \max_a Q(s_{t+n}, a). \tag{2.23}$$

The λ parameter specifies how the above updates can be blended together. Suppose an n-step update (essentially the right-hand side of the above update rules) is denoted as U_n. Then a blended update may be specified as:

$$Q(s_t, a_t) \leftarrow \frac{1}{2}U_1 + \frac{1}{4}U_2 + \frac{1}{4}U_3. \tag{2.24}$$

The only requirement is that the weights used for different updates add up to unity. The TD(λ) approach is to decay the weights as n grows, so that the weight of the nth update is:

$$w(n) = (1 - \lambda)\lambda^{n-1}. \tag{2.25}$$

Thus, the one-step updating rule of Q-learning is obtained by assigning (by convention) $\lambda = 0$. TD(λ) has been shown to outperform Q-learning for many environments when the right choice of $1 > \lambda > 0$ can be determined. Most of the initial theoretic results about TD are due to Sutton [72] and Dayan [62].

2.4.1.8 Generalizing Learned State Values

The Q-learning algorithm is based on the notion of a table that maps state-action pairs to values. When the algorithm encounters a state that it did not see before, its decisions are likely to be highly ineffective[17] despite the fact that very "similar" states might have been encountered many times in the past.

The problem with the tabular approach is that there is no notion of "similarity" of states, which would allow us to ignore details of the state that are irrelevant to the immediate decision that has to be made. In other words, the pure tabular approaches lack the ability to *generalize*. Such an ability is especially important in learning tasks with a very large number of states and actions. Even if we had all the memory necessary to represent every state-action pair in the table, we would also need the sufficiently long and varied experience to fill the elements of such a table.

How can we, then, apply reinforcement learning to tasks where most of the states encountered would not have been seen before in their exactness? The answer is to move away from the exact tabular representation and apply a generalization technique, many of which have already been studied in Machine Learning, particularly, in *supervised* learning. Examples of such techniques include those used in artificial neural networks, statistical pattern recognition, and Bayesian learning.

If we are no longer storing the exact Q values in a table, we can instead use some parameterized function with a set of parameters θ_i forming a vector $\boldsymbol{\Theta}$. The idea is that this

[17]Essentially random actions are expected since the value of each action in this state is equal, e.g. zero.

function will approximate the optimal Q values for each state-action pair. Instead of updating the Q value directly, we must now adjust Θ with the hope of moving the function value at the current state towards the desired target value obtained from experience. By adjusting Θ, we will also affect the Q values of other states, possibly making them diverge from the optimum. This, however, is a natural and expected consequence of using a function approximator. After all, we are using less memory by incorporating the function approximator, as compared to storing the entire Q table, or even the Q table only for the observed states.

What function approximation techniques can we use for RL? Bertsekas and Tsitsiklis [19] have presented many function approximation methods. While a variety of such techniques exist, the interactive nature of the RL tasks imposes a preference towards incremental methods able to incorporate new information as the interaction progresses. As pointed out in [81, p.195], RL additionally requires function approximation to support target functions that change as the policy changes.

We would like to choose function approximation methods that best minimize mean-squared error (MSE) over some distribution of inputs, which in the case of state-value estimation (the so-called *prediction* problem) are the different states visited by the RL agent. Thus, the MSE for estimating $V^\pi(s)$ using Θ is:

$$\mathrm{MSE}(\Theta) = \sum_{s \in \mathcal{S}} \mathrm{P}(s)[V^\pi(s) - V_t(s)]^2.$$

Since it is likely to be impossible to reduce the error at all states (there are many more states than the set of parameter values, θ_i), we may prefer to choose $P(s)$ to correspond to the "training" examples, namely, the visited states during learning in which the updates are performed. In other words, we may not care so much about the states that we are unlikely to visit given the current policy that we are pursuing.

In the most general case we do not expect to reach the globally optimal function approximation such that $MSE(\Theta^*) \leq MSE(\Theta)$, for all possible Θ, where Θ^* is the parameter vector representing the optimal approximation. Such a global optimum may be attainable

in particular cases (e.g. linear approximator), but in the most general case, we expect that a local optimum is obtained, where $MSE(\Theta^*) \leq MSE(\Theta)$, for values of Θ in some vicinity of Θ^*. Unfortunately, in many cases where function approximators are used, convergence to even a local optimum, let alone the global one, cannot be guaranteed with the current state-of-the-art. Moreover, some methods exhibit total divergence with MSE potentially reaching "infinity" as time increases indefinitely. One of the key reasons for this hurdle is the fact that the algorithm needs to chase a moving target. As the value function is being estimated using a function approximation scheme, that function itself changes as a result of the gradual policy change caused by the on-going policy improvement. If the target function changes too quickly no convergence can ever be expected.

One category of techniques for MSE minimization suitable for RL are the so-called *gradient-descent* methods. In gradient-descent methods, the function being approximated must be a smooth differentiable function of Θ. If we are trying to approximate the state value function $V^\pi(s)$, upon observing a new example, we can make a small adjustment of the parameter vector so that the error is reduced for this particular example:

$$\Theta_{t+1} = \Theta_t - \frac{1}{2}\alpha\nabla_\Theta[V^\pi(s_t) - V_t(s_t)]^2 = \Theta_t + \alpha[V^\pi(s_t) - V_t(s_t)]\nabla_\Theta V_t(s_t),$$

where α is the usual learning rate parameter, and $\nabla_\Theta V_t(s_t)$ is the gradient of V with respect to Θ, i.e. it denotes the vector of partial derivatives of V. To guarantee that such a gradient-descent method converges, the step-size parameter α must be gradually reduced.

As mentioned earlier, linear function approximators may be particularly attractive due to their tractable analytical properties, although other approximators exist. In particular, the *coarse* and *tile*[18] coding approaches compose the function out of the so-called *features*. Kanerva coding [36] is also used where binary features correspond to particular *prototype* states. Finally, in the case of a Radial Basis Functions approach [21], the binary features

[18]Better known in the literature as the "cerebellar model articulator controller" (or CMAC) which was introduced by Albus in [4] and [5].

are extended and become continuous-valued. Choosing these features carefully allows one to add prior domain knowledge to RL systems.

Research in this area is important because most practical RL problems necessarily require some form of generalization due to the enormity of the state space. So far, convergence proofs of methods discussed here have been elusive and only possible in special cases. Theoretical research, therefore, has concentrated on state-value approximation rather than on the complete RL problem of finding the optimal policy.

2.4.1.9 Directions of Future Research

By no means does this complete the review of the RL algorithms or even the types of algorithms that are available. For instance, we omitted from our discussion the *model-based*[19] algorithms that do not have to relearn the basic environment dynamics because of a change in the task that they are asked to perform. There also exist the *Partially Observable* MDP (or POMDP) algorithms that do not make the assumption that the current environment state is available, in its completeness, to the agent. Except for the multi-automata schemes discussed in the context of Learning Automata (see Section 2.4.2.6), we also omitted from this review systems consisting of multiple interacting agents (i.e. *multi-agent* systems) which have been recently studied to assess how collective intelligence can emerge from simple cooperating agents.

While paying homage to the foresight of Turing (quoted at the beginning of this section), Nilsson [54] also points out what the Reinforcement Learning research community needs to consider in order to successfully incorporate the learning capability into an intelligent (or what Nilsson calls "habile") system:

> *At some stage in the development of a habile system, its reinforcement learning mechanisms must be able to work with the perceptual, representation, and*

[19] As opposed to model-free approach of Q-learning.

action hierarchies ... and with relational descriptions of states. These remain important areas for future research.

Nilsson cites [10] and [73] as two pieces of existing work in this direction.

2.4.2 Learning Automata

Learning Automata (LA) is a sister field to RL, and this survey would not be complete without including results from the it. It is also relevant to this Thesis, because the results pertaining to the *discretization* of LA algorithms serves as an inspiration for our attempts at discretizing Q-learning (see Section 3.4.3.5).

2.4.2.1 LA Model

The classical feedback loop in the Learning Automata (LA) [53, p.35] field is even simpler than that of the RL scheme (but see Section 2.4.2.6). Figure 2.8 shows an automaton selecting an action $\alpha \in \{\alpha_1, \alpha_2, \ldots \alpha_r\}$, and the Environment producing a response β.

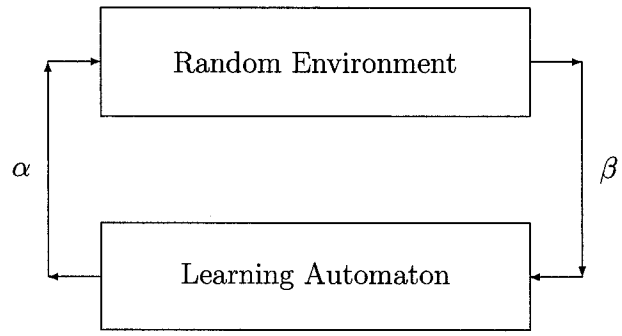

Figure 2.8: Classical LA scheme

The number of possible actions is usually assumed to be finite [86, p.10]. In the simplest case, β is either 0 (representing a favourable response, or a reward) or 1 (representing an

unfavourable response, or a penalty). This is known as the P-model. Alternatively, in the Q-model, β can take on a finite number of real values $\{\beta_1, \beta_2, \ldots, \beta_q\}$, where $\beta_i \in [0, 1]$. Finally, in the S-model, β can be any real number in $[0, 1]$ (see [53, p.39] for discussion of these models).

While the response of the Environment is analogous to the reinforcement signal in RL, in the classical LA model the current state of the Environment is not communicated to the automaton. This happens because the environment never changes states, i.e. it is assumed to be stationary. We should point out, however, that more complex LA schemes have also been studied. For instance, one way in which a non-stationary environment can be dealt with is to assume that the latter can be modeled by multiple stationary environments, which switch according to a Markov process [53, p.230]. An underlying ergodic automaton is then used so as to never fully converge to a single probability distribution. The faster the Environments switch, the fewer states the automaton will have in order to facilitate a faster adaptation to the new environment. The problem with such an adaptation to non-stationary environments is that delayed rewards/penalties are not considered, and therefore, the overall automaton is likely to follow, in a greedy fashion, a suboptimal policy. Other LA schemes do allow the environment to provide the automaton the so-called *context vector* [86, p.33] which effectively represents the current state of the Environment. Networks of LA (see Section 2.4.2.6) can also be assembled to interact with a single so-called G-Environment[20] [86, p.107] where the entire network receives the context vector.

A stationary Environment is assumed to be governed by a probability distribution (a vector of probabilities in the P-model) over the actions of the automaton. The Environment chooses its response in a random fashion based on the action of the automaton, and the penalty probability for this action. Various schemes are known to successfully converge to the optimal action with various degrees of learning capability. More specifically, an

[20] G stands for Generalized.

automaton is called *expedient* if it tends to behave better than the pure-chance automaton which assigns equal probabilities to all actions. An optimal automaton always converges to the optimal action having the least penalty probability — unfortunately, no such automata exist! An automaton is said to be ϵ-optimal if it can get closer than ϵ to the smallest expected penalty, and this quantity ϵ can be made as small as necessary by changing some automaton parameter. More formal definitions of expediency and ϵ-optimality are to follow.

2.4.2.2 Fixed-Structure Stochastic Automata

The so-called *Fixed-Structure* Stochastic Automata (FSSA) [53, ch.3] have a fixed number of possible states that the automaton can be in. For each such state, fixed action selection probabilities are assumed. The state of an FSSA changes based on the previous state, and thus, action, as well as the response of the Environment. The state transition probabilities are also fixed. Many FSSA can be further characterized as *deterministic* because both their state-transition function and their action-selection function are deterministic.

FSSA were the very first Learning Automata introduced by Tsetlin in his seminal paper [87], and by his followers Krinsky [40], Krylov [41], Ponomarev [64], etc., all of whom have different FSSA attributed to them. We present here the original Tsetlin automaton as it is among the simplest to discuss and analyze. This automaton has only two states ϕ_1 and ϕ_2 as well as the two corresponding actions α_1 and α_2. Such an automaton is customarily labeled as $L_{2,2}$. The Environment response is confined to $\beta = 0$ (reward) and $\beta = 1$ (penalty), in other words, we are dealing with the P-model. This arrangement is also analyzed in the literature using the Two-Armed Bandit Problem, which was studied extensively since the early 1950s [69][81, ch.2][53, p.101]. Figure 2.9 shows the state-transition and action-output diagram for the Tsetlin automaton given the two different responses of the environment.

When the automaton is in state ϕ_1 it always chooses action α_1, and, likewise, when in state ϕ_2 it always chooses action α_2. Recall that the Environment's responses are based on

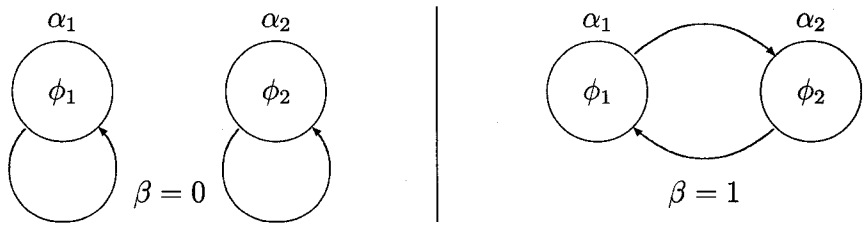

Figure 2.9: The $L_{2,2}$ Tsetlin automaton

fixed probabilities $c_1 = Pr(\beta = 1|\alpha = \alpha_1)$, and $c_2 = Pr(\beta = 1|\alpha = \alpha_2)$. Suppose now that $c_1 > c_2$, i.e. the likelihood of penalty on action α_1 is greater than on action α_2. Suppose also that the automaton is in state ϕ_1, thus, choosing action α_1. If the Environment responds with a penalty, according to the Figure, the automaton will switch its state to ϕ_2, i.e. the state with smaller probability of penalty. Of course, the automaton will switch the state back to ϕ_1 if a penalty is produced on action α_2 (for, after all, in the most general case $c_2 > 0$, even though $c_1 > c_2$), but since $c_2 < c_1$, the automaton is expected to spend more time in state ϕ_2 and, thus, favour the better action α_2. The above rather informal reasoning can be formalized by introducing the definition of expediency.

Let us first define a quantity $M(n)$ as the average penalty for a given action probability vector at time step n:

$$\begin{aligned} M(n) &= \mathrm{E}[\beta(n)|p(n)] = \Pr[\beta(n) = 1|p(n)] \\ &= \sum_{i=1}^{r} \Pr[\beta(n) = 1|\alpha(n) = \alpha_i] \Pr[\alpha(n) = \alpha_i] \\ &= \sum_{i=1}^{r} c_i p_i(n), \end{aligned} \tag{2.26}$$

where $p_i(n)$ is the probability of selecting action α_i at time step n. For example, for a pure chance automaton, $p_i(n)$ is constant and equal to $\frac{1}{r}$, where r is the total number of actions. Since we are examining the case of $r = 2$, the pure chance automaton has $M(n) = M_0 = \frac{c_1+c_2}{2}$. For an automaton to do better than pure chance, the average penalty

should be lower than M_0 at least asymptotically as $n \to \infty$. We are now ready to formally define expediency.

Definition 2.1 *A learning automaton is said to be* expedient *if:*

$$\lim_{n\to\infty} \mathrm{E}[M(n)] < M_0.$$

In order to determine expediency of the $L_{2,2}$ automaton, we need to establish the limiting probabilities π_1 and π_2 of two actions, namely:

$$\pi_i = \lim_{n\to\infty} p_i(n).$$

Since the environment dynamics are described by two constant penalty probabilities c_1 and c_2, the evolution of the automaton is governed by an ergodic Markov chain with the state transition matrix P being:

$$P = \begin{bmatrix} 1-c_1 & c_1 \\ c_2 & 1-c_2 \end{bmatrix}.$$

The final probabilities of two states ϕ_1 and ϕ_2 (and consequently of two actions) can be obtained by solving the following equation:

$$P^T \vec{\pi} = \vec{\pi}, \tag{2.27}$$

where $\vec{\pi}$ is a vector of probabilities π_i introduced earlier, and T is the matrix transposition operator. By adding the full probability constraint $\pi_1 + \pi_2 = 1$, and solving this equation, we obtain:

$$\pi_1 = \frac{c_2}{c_1 + c_2}, \quad \pi_2 = \frac{c_1}{c_1 + c_2}.$$

We, therefore, get:

$$\lim_{n\to\infty} M(n) = \sum_{i=1}^{2} c_i \pi_i = \frac{2c_1c_2}{c_1 + c_2}.$$

When $c_1 \neq c_2$,

$$\frac{2c_1c_2}{c_1 + c_2} < \frac{c_1 + c_2}{2},$$

and hence, except for the trivial case when the two penalty probabilities are the same (i.e. no action preference exists), the Tsetlin $L_{2,2}$ automaton is shown to be expedient.

Let us now briefly consider other criteria (besides expediency), which allow us to assess and compare the qualities of LA, namely, the *optimality* and *ϵ-optimality*. In [53, p.54], these are collectively referred to as *norms of behavior*.

Definition 2.2 *A learning automaton is said to be* optimal *if:*

$$\lim_{n\to\infty} \mathrm{E}[M(n)] = c_\ell,$$

where c_ℓ is the smallest of the penalty probabilities, i.e. $c_\ell = min_i\{c_i\}$. In other words, an optimal automaton will asymptotically always (i.e. with unity probability) choose the optimal action, that is, the action with the smallest probability of resulting in a penalty.

Optimality, while clearly the desired property for an automaton, is, in practice, impossible to attain. It is often more practical to require that the automaton be somewhat less optimal, leading to the following definition.

Definition 2.3 *A learning automaton is said to be* ϵ-optimal *if:*

$$\lim_{n\to\infty} \mathrm{E}[M(n)] < c_\ell + \epsilon$$

can be obtained for any arbitrary $\epsilon > 0$ by an appropriate choice of the parameters of the automaton.

Note that other norms of behavior, such as *absolute expediency* also exist.

The Markov chain based analysis presented here is characteristic of many FSSA, some of which can be shown [53, ch.3] to be not only expedient but also absolutely expedient and/or ϵ-optimal. The norms of behavior as defined above are also applicable to variable-structure automata described in the following Section. Successful applications of FSSA include solutions to the *equipartitioning* problem [61].

2.4.2.3 Variable Structure Stochastic Automata

The family of *Variable-Structure* Stochastic Automata (VSSA) [53, ch.4] can change both their action selection probabilities and state transition probabilities at every time step during the learning process. This characteristic enables VSSA to achieve better convergence results. Varshavskii and Vorontsova [90] were the first to suggest automata that update transition probabilities. Rather than enumerating all states of the LA as an explicit set $\phi_1, \phi_2, \ldots, \phi_n$, the VSSA utilize a set of state variables. Usually these are real-valued variables and many of the well studied LA choose to represent the action probabilities through them. If these action probabilities are updated using linear equations, the resulting reinforcement scheme is also called linear. Linear schemes are among the best studied due to their susceptibility to analysis. We introduce here the linear reward-penalty (L_{R-P}) and linear reward-inaction (L_{R-I}) as the two most prominent schemes. These schemes possess interesting and significantly different characteristics due to which they serve as prototypes for distinct types of behaviour observed in all LA. For the sake of simplicity, we present the linear schemes for automata with only two actions.

Linear Reward-Penalty

Let $p_i(n)$ be the probability of choosing action α_i at time step n. Then the evolution of the L_{R-P} automaton can be specified by:

$$\left.\begin{array}{rcl} p_1(n+1) & = & p_1(n) + a(1 - p_1(n)) \\ p_2(n+1) & = & (1-a)p_2(n) \end{array}\right\} \quad \alpha(n) = \alpha_1, \beta(n) = 0$$

$$\left.\begin{array}{rcl} p_1(n+1) & = & (1-a)p_1(n) \\ p_2(n+1) & = & p_2(n) + a(1 - p_2(n)) \end{array}\right\} \quad \alpha(n) = \alpha_1, \beta(n) = 1.$$

Here a is a reward/penalty parameter such that $0 < a < 1$. We can observe that in a two-action case $p_1(n) = 1 - p_2(n)$ and so the above updating rules can be rewritten in terms

of p_1 only:

$$\begin{array}{lll} p_1(n+1) = p_1(n) + a(1-p_1(n)) & \alpha(n) = \alpha_1 & \beta(n) = 0 \\ p_1(n+1) = (1-a)p_1(n) & \alpha(n) = \alpha_1 & \beta(n) = 1 \\ p_1(n+1) = (1-a)p_1(n) & \alpha(n) = \alpha_2 & \beta(n) = 0 \\ p_1(n+1) = p_1(n) + a(1-p_1(n)) & \alpha(n) = \alpha_2 & \beta(n) = 1. \end{array} \tag{2.28}$$

In other words, the L_{R-P} automaton updates probabilities in the same exact way whether a reward is received on action α_1 or a penalty is received on action α_2. From Equation (2.28), the sequence $\{p(n)\}$ can be described as a Markov process whose state space is in the unit interval $[0, 1]$.

It can be shown that for the L_{R-P} scheme:

$$\lim_{n\to\infty} \mathrm{E}[p_1(n)] = \frac{c_2}{c_1 + c_2} \quad \text{and} \quad \lim_{n\to\infty} \mathrm{E}[p_2(n)] = \frac{c_1}{c_1 + c_2}.$$

Hence, we compute:

$$\begin{aligned} \lim_{n\to\infty} \mathrm{E}[M(n)] &= \sum_{i=1}^{2} c_i \mathrm{E}[p_i(n)] \\ &= c_1 \mathrm{E}[p_1(n)] + c_2 \mathrm{E}[p_2(n)] \\ &= \frac{2c_1c_2}{c_1 + c_2}. \end{aligned} \tag{2.29}$$

Note that this is exactly the same value that was obtained for the Tsetlin $L_{2,2}$ automaton. The L_{R-P} automaton is, therefore, also expedient. Unlike the L_{R-I} scheme (see following Section), L_{R-P} cannot be ϵ-optimal. To achieve ϵ-optimality, a modification called $L_{R-\epsilon P}$ was proposed (see [53, p.114]).

Linear Reward-Inaction

Let us now consider a simple modification of the probability update rules of Equation (2.28). This time, the probabilities will not be updated (i.e. they must be left the same) if a penalty

is received. Otherwise, we will do what Equation (2.28) prescribes. The resulting set of update rules is summarized below:

$$\begin{array}{lll} p_1(n+1) = p_1(n) + a(1 - p_1(n)) & \alpha(n) = \alpha_1 & \beta(n) = 0 \\ p_1(n+1) = p_1(n) & \alpha(n) = \alpha_1 & \beta(n) = 1 \\ p_1(n+1) = (1-a)p_1(n) & \alpha(n) = \alpha_2 & \beta(n) = 0 \\ p_1(n+1) = p_1(n) & \alpha(n) = \alpha_2 & \beta(n) = 1. \end{array} \tag{2.30}$$

This scheme is precisely what we referred to earlier as the linear reward-inaction, or L_{R-I} scheme. It was introduced in [76]. Despite the superficial simplicity of the nature of the applied modification, this scheme has a dramatically different behaviour asymptotically. Observe, for example, that now there exist two absorbing states $p_1 = 0$ and $p_1 = 1$. Once the automaton settles into either of these two states it will never "escape" them. For example, when $p_1 = 0$, that is, the automaton always selects action α_2, the state can potentially change only upon receiving a reward ($\beta = 0$) but in this case the expression $(1-a)p_1(n)$ is actually equal to 0 since $p_1 = 0$. We have thus confirmed that $p_1 = 0$ is an absorbing state.

The analysis of this algorithm shows that it is both ϵ-optimal and absolutely expedient [86, p.15], making the L_{R-I} a rather attractive algorithm, especially given its simplicity. Upon further analysis (see [53] for example), it turns out that $p_1(n)$ is guaranteed to converge (i.e. converges with unity probability) to either of the two absorbing states, regardless of what the starting probability $p_1(0)$ is and what the values of c_1 and c_2 are. Of course, we would like to see the convergence to 0 when $c_1 > c_2$ (i.e. α_2 is the best action), and vice versa, a convergence to 1 when $c_2 > c_1$. While this ideal outcome is not guaranteed, it can be shown that under L_{R-I} and when $p_1(0) = \frac{1}{2}$, p_1 can be made to converge to 0 when α_2 is the best action with a probability as close to unity as desired, and vice versa.

The L_{R-P} and L_{R-I} schemes are among the better understood ones. Other types of Learning Automata include the estimator and discretized families examined in the following sections.

2.4.2.4 Pursuit Algorithm

The VSSA described so far updated their action probabilities based only on the last selected action and the instantaneous reinforcement received. In other words, the past history of the selected actions and received reinforcements is not explicitly represented; rather, the current action probabilities indirectly reflect that history. Thathachar and Sastry [85] introduced a new family of algorithms called *estimator* algorithms which successfully utilize the entire history of Automaton-Environment interaction. The most prominent estimator algorithm is the Pursuit algorithm that we describe below.

The key idea of the Pursuit algorithm is to maintain, in addition to the action probability vector, two additional vectors $\vec{Z}(k)$ and $\vec{\eta}(k)$. The elements η_i of the latter vector simply contain the number of times action α_i was chosen until time step k. The elements Z_i, in turn, give the total reinforcement obtained in response to action α_i until time step k. These two additional vectors are used to compute the estimate of the expected reward for each of the actions. The true expected reward is customarily denoted by d_i, whereas its estimate at time step k is denoted by $\hat{d}_i(k)$. Given the two vectors $\vec{\eta}(k)$ and $\vec{Z}(k)$, the estimates are simply:

$$\hat{d}_i(k) = \frac{Z_i(k)}{\eta_i(k)}, \quad i = 1, \ldots, r.$$

Assuming $\alpha(k) = \alpha_i$, vectors $\vec{\eta}(k)$ and $\vec{Z}(k)$ are updated as follows:

$$\begin{aligned} Z_i(k) &= Z_i(k-1) + (1 - \beta(k)) \\ Z_j(k) &= Z_j(k-1), \forall j \neq i \\ \eta_i(k) &= \eta_i(k-1) + 1 \\ \eta_j(k) &= \eta_j(k-1), \forall j \neq i. \end{aligned}$$

In case of the P-model (recall $\beta \in \{0, 1\}$), $\hat{d}_i(k)$ would essentially be $1 - \hat{c}_i(k)$, i.e. the estimate of the reward probability for action α_i at time k.

It is clear that the estimates maintained are simply the running averages of rewards obtained so far for each of the possible actions. The update of the estimates at every time step, in itself, does not affect the action probabilities, and so they must be changed in some other manner. In fact, according to the Pursuit algorithm, the action probabilities are updated in such a way so as to bias them more towards the action $\alpha_{\ell(k)}$ that currently (at time step k) has the "best" estimated expected reward, i.e.

$$\ell(k) = \text{argmax}_i \hat{d}_i(k).$$

The actual update is accomplished in the exact same linear manner that we saw in the case of the L_{R-I}:

$$\begin{aligned} p_{\ell(k)}(k+1) &= p_{\ell(k)}(k) + a(1 - p_{\ell(k)}(k)) \\ p_j(k+1) &= p_j(k) - ap_j(k), \forall j \neq \ell(k). \end{aligned}$$

Clearly, in the Pursuit algorithm, the updating of action probabilities does not directly depend on the last selected action. In some cases, of course, the choice of an action at step $k-1$ may cause the $\ell(k)$ to change to $\alpha(k-1)$, thus causing the probability of $\alpha_{\ell(k)} = \alpha(k-1)$ to be increased as well.

In general, the Pursuit algorithm is quite successful as it is usually an order of magnitude faster than algorithms such as the L_{R-I}, while retaining the attractive properties of the latter, e.g. ϵ-optimality. The popularity of this algorithm led to many extensions, including generalized pursuit learning schemes [1].

2.4.2.5 Discretizing LA Algorithms

The equations governing the behaviour of the presented LA algorithms are expressed in terms of continuous probability values/vectors which, in practice, are impossible to represent exactly in a computer. Naturally, discretization must occur if these values are represented in some format, such as the IEEE 754 floating-point representation. It turns out that we can

extract some additional benefit if we explicitly consider discretizing the learning algorithms. By doing so, we reduce the memory requirements and improve the speed of convergence — both being profitable ways of expanding the applicability of the algorithms.

Let us now consider how the L_{R-I} scheme can be discretized. The idea is to restrict the action probabilities to a finite number of values in the range $[0, 1]$. For the linear algorithms, allowed values are spaced evenly in this range, usually at a distance $\Delta = \frac{1}{rN}$, where r is the number of actions, and N is the so-called *resolution parameter* specifying the desired level of discretization. According to the L_{R-I} scheme, the action probabilities do not change on penalty, but they do change on reward, in such a way, so as to linearly increase the probability of the rewarded action, and, hence, decrease all other probabilities. Given that the last chosen action was α_i, in the discretized version (abbreviated as DL_{R-I}) the Equation (2.30) is extended to the multi-action case and is replaced by:

$$\begin{aligned} p_j(k+1) &= max\{p_j(k) - \Delta, 0\}, \quad \forall j \neq i \\ p_i(k+1) &= 1 - \textstyle\sum_{j \neq i} p_j(k+1). \end{aligned} \tag{2.31}$$

In other words, we reduce the probabilities of all other actions and thus increase the probability of α_i.

The DL_{R-I} algorithm is known to be generally faster than the traditional continuous L_{R-I} with the speedup occurring when the action probabilities are near the range bounds of 0 and 1 (this is where the continuous algorithm is tedious). The DL_{R-I} algorithm has also been shown to be ϵ-optimal in at least the two-action case. Note that the discretized version of the L_{R-P} also exists [56].

Since the Pursuit algorithm uses essentially the same approach to update its action probabilities as L_{R-I}[21], we can also speak of a discretized version of the Pursuit algorithm. We can still maintain the estimates in the same exact continuous manner, but update the

[21]The difference is that L_{R-I} increases the probability of the executed action, whereas the Pursuit does the same for the action with the best current estimate.

action probabilities as below:

$$\begin{aligned} p_j(k+1) &= max\{p_j(k) - \Delta, 0\}, \quad \forall j \neq \ell \\ p_\ell(k+1) &= 1 - \textstyle\sum_{j \neq \ell} p_j(k+1), \end{aligned} \tag{2.32}$$

where ℓ is again the index of the action with the maximum current estimate. As with the DL_{R-I}, the Discretized Pursuit algorithm is known to be experimentally faster[57] while still being ϵ-optimal with respect to the resolution parameter, N.

The idea of LA with discretized probability is due to Thathachar and Oommen [84]. Since then, many LA have been discretized and analyzed [60] [58] [1] [42].

2.4.2.6 Multi-automata Schemes

A single automaton operating in a stationary environment may not, by itself, be highly useful or even have wide applicability. A collection of such simple automata may, however, be able to tackle more interesting problems. Just like research in the RL domain is interested in multi-agent systems, so does the LA research deal with multi-automata systems.

One possible scheme where multiple automata can be used is one where the individual actions of the participating automata are composed to form an aggregate action. Each automaton may receive an independent reinforcement from the Environment, or the exact same reinforcement may be delivered to all participating automata. The latter model is often referred to as the *common payoff* game. Alternatively (as is the case in the Goore game described below), each automaton may receive an independent reinforcement sampled from the same exact distribution which potentially changes as the interaction progresses.

Goore Game: An interesting example where a multi-automata system is successful at learning in a distributed fashion is the Goore game described by Tsetlin [88]. In this game, several two-action automata simultaneously choose a vote of YES or NO. The proportion y of YES votes determines the probability $f(y)$ with which every automaton is independently

rewarded (the P-model is being assumed, i.e. $\beta \in \{0, 1\}$). If the function $f(y)$ is *unimodal*, i.e. it has a single local maximum, such a collection of automata is guaranteed to converge to the optimal proportion of y (i.e. the proportion for which the probability of reward $f(y)$ is highest) given the sufficient number of participating automata and given that each automaton is *absolutely expedient.*

Feed-forward Network of LA: In the Goore game all automata play an equivalent "symmetric" role. Several schemes have been explored where the automata are instead arranged hierarchically in a fashion similar to the Artificial Neural Networks, and specifically, similar to the feed-forward networks based on the backpropagation of error. Figure 2.10 shows a possible network where the individual nodes have inputs (context vector $\mathcal{X}$) and outputs (actions α), and also receive reinforcements β. Since the same exact reinforcement is delivered to every node, this network falls under the common-payoff category.

The individual nodes in the network are not single automata, but are teams of automata assembled according to Figure 2.11. The context vector is not fed into each of the automata in the team. Instead, it is combined with automata actions according to a fixed function Γ in order to produce the action of the team.

The analysis of such feed-forward networks of LA can be found in [86, p.116]. Such networks have been shown to converge to a local maximum only. The advantage of such networks is that the amount of memory required for the entire network is considerably smaller than that required for a single generalized automaton which accepts context vectors.

2.4.2.7 Relevance to this Thesis

The above review of the LA field has omitted other existing schemes such as the Continuous Action LA (actions are from the continuous domain) or Parameterized LA where the action probability is a parameterized function with the LA learning the parameters.

Despite the relative simplicity of the commonly studied single-automaton LA schemes

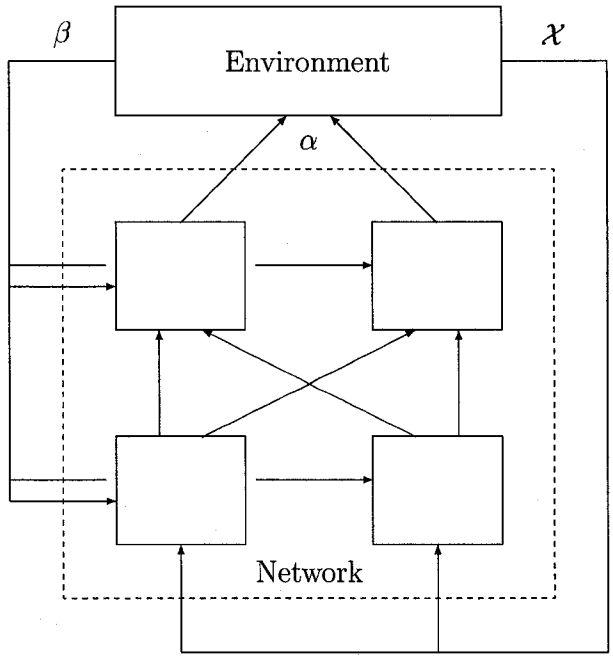

Figure 2.10: A feed-forward network of Learning Automata. Each node corresponds to a team of automata.

(as compared to the general RL problem), this field was historically very influential to RL research. Furthermore, there are many important contributions that can *still* be transferred and adapted to the general paradigm of Interactive Learning and RL, in particular. The fact that the LA setup has a single feedback signal specifically hints at one generalization of the RL scheme that we can attain, namely, that of removing the distinction between the reinforcement signal, and the regular output of the Environment reporting on the current state. Also, the ideas used in discretizing the LA algorithms serve as an inspiration in our Discretized Q-learning introduced in Chapter 3.

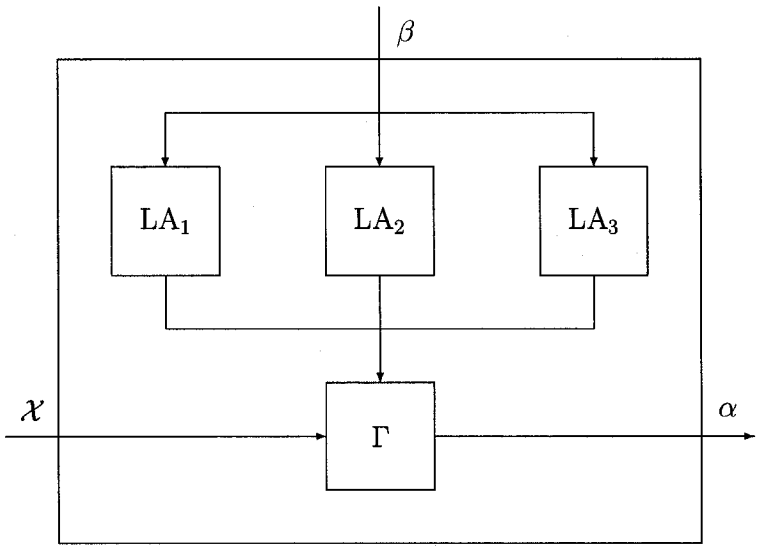

Figure 2.11: A node of the feed-forward network is constructed as a team of LA and a function Γ.

Summary

In this Chapter we have postulated that a system behaving "intelligently" must rely on two interdependent processes of learning and planning. We consequently reviewed the paradigms within which these processes are studied separately. These include Supervised Learning and Planning (or Search for plans). We then presented the review of frameworks where the two processes are brought together in order to construct an "intelligently" behaving system, i.e. Reinforcement Learning, and to a lesser degree Learning Automata, inasmuch as Planning is featured there in a less explicit form. We have paid particular attention to the RL framework, as our own research is based on the generalization of the feedback loop, the interaction process, and the reinforcement signal as a means of supplying the goal information.

We conclude by providing a final quote from Nilsson's article [54], where he encourages work towards the "core" intelligent system capable of automating many human tasks:

> *Rather than work toward this goal of automation by building special-purpose systems, I argue for the development of general-purpose, educable systems that can learn and be taught to perform any of the thousands of jobs that humans can perform. ... I advocate beginning with a system that has minimal, although extensive, built-in capabilities.*

In doing the work reported in the following Chapters, we have adapted the same minimalist approach, attempting, whenever possible, to provide our learning agents very little initial information about the problem domain, and to require them to learn what they *need* to achieve their respective goals.

Chapter 3

Æip — Agent-Environment Interaction Protocol

3.1 Introduction

A scientific approach to the formal understanding of intelligence requires that theories are supported by repeatable experiments independent of the experimenter. This implies a behaviouristic approach to quantification of "intelligence" (or rationality) exhibited by the behaviour of a system in its environment. We, as AI researchers, therefore, need to define the notions of a system and environment and their interaction. As a practical matter, we also need to standardize these notions so as to be able to compare proposed algorithms and assess their merits. Such standardization must accommodate various new developments in the field of interactive learning such as collaborative or competing multi-agent systems. We must also allow for a variety of ways in which agents and environments can be interacting including such possibilities where two agents act as environments *for each other*. These are precisely the reasons motivating a general Agent-Environment interaction protocol (or Æip) described in this chapter. Based on the principles of Æip, we also offer a novel way of cat-

egorizing goals and describe an alternative signal that supplies the goal information to the agents.

3.2 Generalization of the Agent-Environment Interaction Protocol

Our ultimate goal is to treat the agent and the environment as *symmetric entities* with the sole subjective distinction that one entity (namely, the agent) is employing a learning algorithm and is, therefore, likely to exhibit an evolving behavior. Once we start treating agents and environments equally, we are only one generalization step away from introducing a multi-agent framework of interconnected entities, some of which are adaptive (*agents*), while others are purely reactive (*worlds*). Eliminating asymmetries between the agent and the environment proceeds in three stages described in the subsections below.

3.2.1 Actions as Observable States of the Agent

First of all, we observe that in the Reinforcement Learning literature, there is a semantic difference between the way we treat the agent's action and the environment's response. The latter is supposed to reflect on the current stable state of the environment, while the agent's action is something that occurs during a particular time interval, and has all the attributes of an event. In order to remove this semantic asymmetry, we propose to replace the agent's action by the agent's response, which, in turn, has the same semantics as the environment's response — namely that of being the current observable state of the agent! In other words, the agent "observes"[1] the current state of the environment, and responds by changing its own observable state. The environment, in turn, observes the new state of the agent and, in

[1]By observation we mean any form of sensing or perception.

response, changes its own state. By modeling the system in this fashion, the agent and the environment also become symmetric in their treatment of both inputs and outputs.

Let us illustrate how we can translate the event-like actions of a robot into the appropriate changes of its observable state. Suppose that we are experimenting with the classic grid world with a robot capable of moving into one of the adjacent grid cells. In the classical RL scenario, the agent will have 4 actions for moving the robot into one of the four directions: north, south, east or west. If the robot has to move by two cells in the same direction, the agent must explicitly select the same action in two successive iterations. Once we replace the actions by an observable state, the agent will still have four distinct observable states that can be interpreted as different positions of the guiding wheels of the robot. If the robot has to move by two cells in the same direction, the agent must change the position of the guiding wheels once, and simply keep them in the same state for the second iteration, i.e. the change of the agent's observable state has to occur only once.

Till now we have been treating the observable state as a single value taken from S, the set of states. In most applications, however, it is natural to treat the state information as being composed of a number of individual and possibly independent state variables. In a grid world, for example, the exact location of the robot can be represented by two coordinates, which are the row and column indices of the cell that contains the robot. Likewise, the agent's state can be subdivided into two components: the state of the motor (on/off) and the direction of the guiding wheels. We, therefore, make a further generalization by representing the agent's sensors as an array of *inports*, and its observable state as an array of *outports*. Figure 3.1 shows an agent connected to its environment through a number of communication *lines* attached to ports. Note that inports are surrounded by a box to visually distinguish them from outports. Also note that, as shown in Figure 3.1, the agent has an additional course of action, namely, to remain in the same cell by turning its motor off.

Suppose now that we would like to allow for a second robot (controlled by a separate agent) to operate inside the grid. Let us denote the two agents as agent A and agent B.

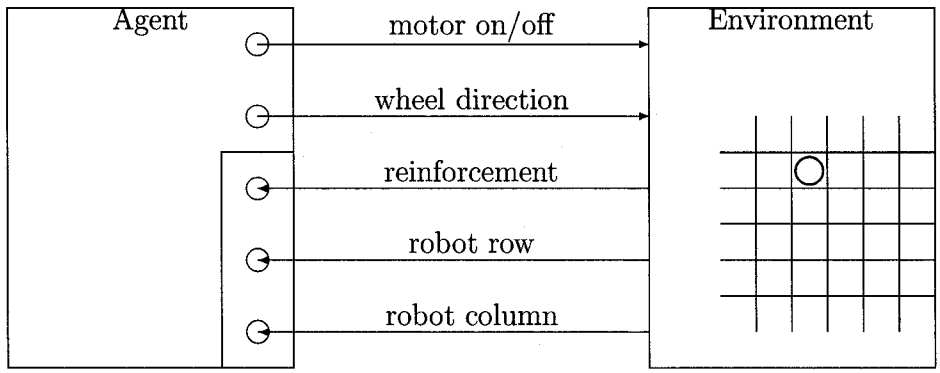

Figure 3.1: An agent connected to its environment through inports and outports

The environment for agent A will include the grid itself plus the agent B. The environment of agent B, on the other hand, includes agent A in addition to the same grid. Figure 3.2 underscores this relative aspect of the environment, and shows that it is no longer convenient to model this scenario using the familiar agent-environment interaction. Instead, it is more appropriate to consider this as an interaction that occurs between 3 components: the two agents and the grid itself. The grid, here, is a non-adaptive component that acts as a *world* within which the agents exist, and which encodes the rules of that world. Observe that the grid component is depicted with 4 inports and 6 outports, which support the two agents. In other words, the grid is very similar to the agents themselves, except for the fact that its behavior is purely reactive and depends solely on the inputs from the two agents. Therefore, instead of dealing with agents and environments as two different concepts, we can speak of them collectively as an ensemble of interacting *entities*.

In order to differentiate between adaptive agent entities and non-adaptive entities such as the grid entity, we refer to the latter as a grid *world*. We are, thus, moving away from the traditional "environment" perspective and terminology.

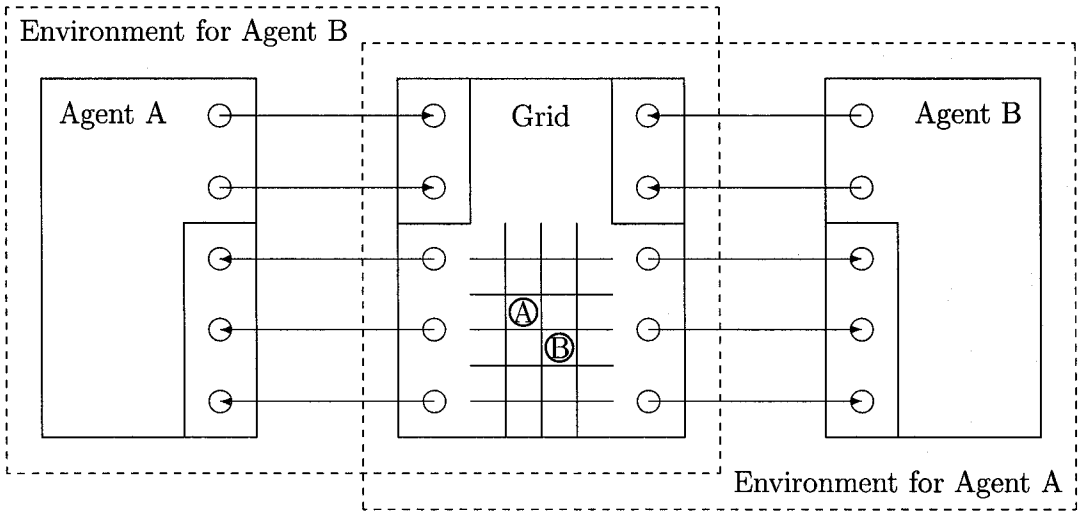

Figure 3.2: Two agents interacting with the same world

3.2.2 Perception

The ports of interacting entities are connected by the unidirectional communication lines through which the values that appear on the outports of one entity are fed into the corresponding inports of another. This transfer of sensory data can occur in one of two ways:

(a) synchronous polling of the communication lines, or

(b) asynchronous interrupting of an interested entity when the signal on the line changes.

From the system design point of view, the first approach is often preferred as it is simpler, and does not require simulated or real concurrency that is needed for interrupts. Systems designed with the polling line disciplines are also easier to analyze and predict, which makes them the choice of real-time system designers. In Æip we want to keep the experimenters' options open, and hence allow for both types of line disciplines. In the discrete-time simulation, however, where the interactions between entities are well orchestrated, polling naturally plays the dominant role.

3.2.3 Reinforcement as Part of the Observable State of the World

Figures 3.1 and 3.2 shown above, demonstrate yet another generalization that we have not discussed so far. In classical RL, the reinforcement signal is markedly different from the environment's response. While the latter is often represented by integer values, reinforcement is always a real-valued signal. Now that we allow multiple inputs and outputs for an entity, it makes sense to consider the reinforcement signal as just another input into the agent. This is exactly what Figure 3.1 depicts. In general, each inport and outport will accept one particular type of data, and the interaction can proceed only if the types of the connected inport/outport pairs match. By doing this, the reinforcement signal is just one particular kind of a signal, which, by convention, always corresponds to the *first* inport of the agent. It is also appropriate that a reinforcement signal usually originates from a world entity, because the determination of whether the agent's actions lead to a solution can only be based on the internal state of the world. Our framework, however, does not impose any restrictions on the source of the reinforcements, in general. Furthermore, this generalization suggests that there might be alternative ways of supplying the goal information to the agent. These are discussed in Section 3.3 below.

3.2.4 Scheduling of Entities' Interactions

We must finally consider how the provision for an ensemble of interconnected entities affects the interaction process itself. In the classical RL scheme, we have an alternating agent-environment operation that must now be extended to an ensemble of interacting entities. Since we have practically eliminated all distinctions between the agent and world entities, we achieve this by requiring that all entities simultaneously update their state based on their currently observable states. While this idea is very attractive, as it removes a further source of asymmetry between agents and environments, it presents implementation challenges when we try to adapt the existing RL algorithms (such as Q-learning) to Æip. We report on these

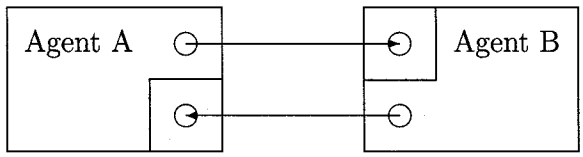

Figure 3.3: Two agents acting as environments for each other

challenges in Chapter 4.

As an alternative to the simultaneous updating of all the entities, we could also require a *universal simulation clock*, and an indication of whether a given entity will be updating itself at each clock *tick* or at a slower rate (e.g. on every 3^{rd} tick). One can, thus, obtain a collection of interacting entities that evolve at various speeds, and which are not necessarily in synchronization with each other. This naturally extends to a continuous-time case.

3.2.5 Æip and JAGUAR

In this section we have outlined a general-purpose platform for experimenting with adaptive agents using RL algorithms. This platform is a generalization of the classical RL scheme and, for historical reasons [13], is called Æip, which stands for Agent-Environment Interaction Protocol. A Java implementation of this platform, which can be used to conduct experiments, is called JAGUAR — an acronym for Java AGents Unified ARchitecture. See Appendix D for a more detailed account of JAGUAR.

Æip allows for various interesting experiments to be set up. Figure 3.3, for example, shows two agents acting directly as an environment for each other putting the two agents in direct competition. This arrangement allows us to directly compare the performance of the two agents by determining which agent is faster at learning the weaknesses of another, and utilizing this knowledge to defeat the opponent.

As another twist on the multi-agent architecture, consider a simple grid world with a single robot moving from cell to cell. One could imagine an experiment where two or

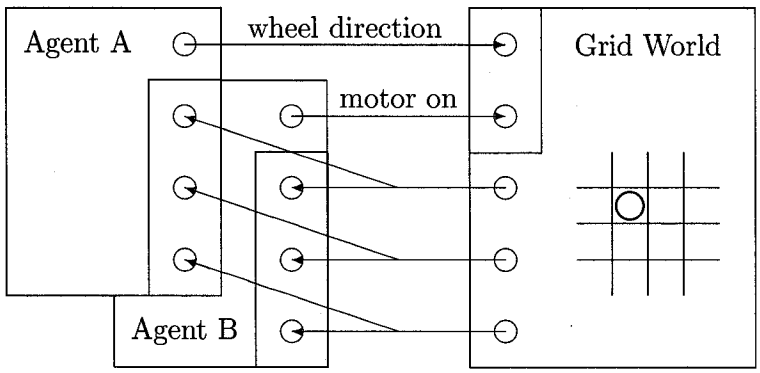

Figure 3.4: Two agents controlling different aspects of the same robot

more agents are controlling different aspects of the same robot. For example, one agent is responsible for choosing the direction, and the other, for selecting forward or backward motion. We conduct this very experiment in Chapter 6. Furthermore, such agents could be in competition , which would give rise to a situation where literally "one arm does not know what the other is doing". The Æip diagram for this curious potential experiment is shown in Figure 3.4.

3.3 Specifying Goals to Agents

3.3.1 Reinforcement Functions

In RL, the goal is supplied implicitly in the form of a sequence of reinforcements (rewards or penalties) where a new reinforcement is given for every action that the agent performs. Thus, in addition to perceptual information from the environment, the agent also receives a special reinforcement signal. A mapping from a previous agent-environment interaction to a particular reinforcement value is called a *reinforcement function.* Experimenters, therefore, are required to design a reinforcement function to be used in a given experiment, and must

ensure that this function corresponds to the desired goal.

Under this scheme, the goal is achieved by maximizing the sum of all reinforcements that the agent receives over its lifetime. We can name two clear advantages of such a measure of the agents' performance. First, the idea of training based on rewards and penalties is well established in psychology, and is employed to teach animals and people alike. It corresponds to our notion of rational behavior driven by the search for "happiness", or general sense of "well-being". We can, thus, make machine learning be analogous to animal learning and, in doing so, draw on the wealth of previous studies in animal and human learning. Second, an individual reinforcement is represented by a single real number, which makes it possible to mathematically analyze the convergence of algorithms, and to employ mathematical schemes such as the discounting of rewards to ensure that their sum stays finite even in non-episodic problems.

There exist, however, several shortcomings of the scheme used in RL. Among the popular algorithms are the so called "model-free" algorithms (e.g. Q-learning) that learn to predict the reinforcement signal, but are not able to predict the next response of the environment. This is analogous to us expecting to become satisfied after drinking a glass of juice, yet not being able to predict that the glass would be empty as a result. Such algorithms are model-free in the sense that they don't learn the model of the environment. While having clear memory advantages, they suffer from being task-specific. This means that the algorithms will have to start learning from scratch, even if the goal is slightly modified.

A much more serious problem of the RL reinforcement scheme, however, has to do with the fact that choosing the correct reinforcement function is often an art rather than a science. As designers of the task, we can recognize desired behavior of the agent, but the choice of which actions are to be rewarded and which to be penalized is often error-prone. In the best case, by choosing an incorrect reinforcement function, we will bias the agent towards a known solution rather than letting it discover a solution on its own. In the worst case, however, the agent will find a way of maximizing the sum of rewards without reaching the

desired goal. If in a game of chess, for example, we reward the capture of opponent's pieces, it may be more beneficial for the agent to lose a game when the opponent has only two pieces left.

Here we propose an alternative method[2] of supplying the goal information to the agent, and compare it with the way it is done in RL.

3.4 Understanding Goals

3.4.1 Different Types of Goals

Goals that we assign to our agents take on a variety of forms. In the simplest case, a goal is a single state of the environment that the agent must reach. Finding an exit from a labyrinth, or solving the Rubik's Cube puzzle are examples of such goals. Equally simple are goals that require the agent to reach one of the *several* desirable states of the environment. Examples of such goals include winning board games like chess or tic-tac-toe. The set of acceptable states can be described explicitly or using a predicate over the environment state space. The path that leads to the goal state can be unconstrained, or may be required to satisfy certain conditions. The well-known problem of a peasant-wolf-goat-cabbage[3] is an example of a problem where the path to goal must not pass through certain "danger" states (e.g., leaving a wolf and a goat unattended).

All of the examples mentioned so far assume the episodic nature of the agent-environment interaction. In other words, the interaction is eventually brought to a conclusion (the game is won or lost, the puzzle is solved, etc.). There is another non-episodic class of problems where the agent continuously interacts with its environment without a natural termination condition. An agent controlling a certain parameter of a manufacturing plant operates

[2]Most of what follows in this section has already been published in [12].

[3]A variant of this problem involves a farmer, a fox, a chicken, and a sack of grain (often corn).

continuously in order to maintain the said parameter within an acceptable range. The classical example of a pole balancing task [81, p.59] also falls into this category[4].

3.4.2 Generalizing Goals

We can generalize the notion of a goal by describing it as an *acceptable trajectory* through the environment state space. If there is more than one such acceptable trajectory, the goal will be described by a set of such trajectories. All tasks[5] that require reaching a certain state (or one of several acceptable states) can be described as a set of all trajectories that pass through the goal state(s). Likewise, a goal of the peasant-wolf-goat-cabbage task can be described as a set of all trajectories that eventually pass through the goal state (peasant, wolf, goat, and cabbage are on the other bank of the river) avoiding all danger states. In the non-episodic task of plant control, the goal is specified as a set of trajectories which consist only of states where the controlled parameter is within the acceptable range.

In the most general case, we might be interested in a particular sequence of actions that the agent performs, and not merely in the sequence of *resulting states* of the environment. Given our Æip treatment of actions as observable states of the agent entities, we can now talk of the *state of the universe*, which is the combined state of all the entities involved in an interaction. Thus, we can define a goal as a set of acceptable trajectories through the state space of the universe. Such a definition of a goal allows us to specify the required agent's behavior and the outcome of this behavior. Because of the way we will be specifying goals to agents (see Section 3.4.3.1), it will not be important, from the agent's perspective, to know whether the goal is a trajectory over just the environment state space or over the universe

[4]The same source also suggests that this task can be treated as episodic. We define a more precise notion of episodicity in section 3.4.3.4.

[5]In this discussion we will be using the terms *task* and *goal* interchangeably, *goal* being the preferred term.

state space (which includes the state of the agent itself).

3.4.3 Binary Goals

In the previous discussion, we separated all trajectories into two categories: acceptable and unacceptable. This corresponds to the idea of an agent either succeeding or failing to reach the goal. There seems to be no avenue for any other outcome, despite the fact that many learning tasks require us to differentiate among the many possible "acceptable" solutions. In a game of chess, for example, we prefer the agent to win sooner than later, thus biasing the agent towards shorter winning trajectories. In Section 3.4.4 we shall show how to generalize the notion of acceptability to include such preferential treatment of acceptable trajectories. We begin, however, by examining the simplest case where all acceptable trajectories are treated equally. Goals representing such sets of acceptable trajectories will be termed *binary*.

3.4.3.1 Imparting Goals to Agents

How can we specify to an agent, in the most general way, the task that it is expected to perform? We could explicitly provide an exhaustive list of all acceptable trajectories. Each trajectory can be described as a sequence of state descriptions. Each description can be in exactly the same format that is presented to the agent during its interaction with the environment (i.e. a new language of state descriptions does not need to be devised). This way of specifying goals is attractive because of its simplicity and generality but it fails, of course, as soon as we start dealing with infinite trajectories or infinite-cardinality sets of acceptable trajectories. For example, while the individual trajectories leading to a solution of the Rubik's Cube puzzle are all finite, the number of possible solutions is clearly infinite[6].

To deal with "infinity" we could devise a language that would allow us to use finite length expressions to describe many infinite length trajectories, or infinite sets of such trajectories.

[6]Here by solution we mean *any* path to goal, not just the shortest.

A UNIX regular expression `a.*b$` describes a set of all strings that contain letter `a` and end with letter `b`. A similar language can be created to describe state trajectories[7]. While not as simplistic as before, this approach suffers from several fundamental shortcomings. The language in which the goal information is described must be understandable by all agents. To this end, we must either standardize the language or choose to describe the goal in a language which is already used by the given agent internally to represent knowledge. The latter may not be possible in cases where the agent's internal representation is not easily understandable (e.g. in neural networks), and in cases where the agent must start with no prior knowledge about the environment[8]. Standardizing a language is also quite problematic as it is theoretically impossible to devise a single language that will give finite descriptions to all possible infinite trajectories, or to all possible infinite-cardinality sets of such trajectories.

The two methods presented above attempt to supply the goal information to the agent before the interaction with the environment begins. Since neither seem to offer a solution, we turn to methods where the agent must *learn* what the goal is while it interacts with its environment. Indeed, this very approach has already been successfully used in the RL framework. In the case of *binary goals*, the main idea is that we must inform the agent when it has veered off the acceptable trajectory as soon as we can recognize this fact.

One interesting property of acceptable trajectories is that once you move off it, there is no way to get back on, i.e. every extension of an unacceptable partial trajectory is also

[7]During his graduate work, as part of a course on *Search Techniques* at the University of Ottawa, the author also experimented with alternative ways of supplying goals to search algorithms. Instead of providing a description for the goal state(s), we showed that the goal could also be defined as a trajectory through the state space, rather than just the terminal state. Constraints on trajectories were specified using PADLA (PAth Description LAnguage), reminiscent of regular expressions. While these ideas are still unpublished, they served as inspiration for our work on goal arity.

[8]More generally, the language of internal representation may not be rich enough to describe the desired goal.

unacceptable. This means that the agent receives feedback only when we know that, given the current state of the environment, it will be impossible for the agent to satisfy the goal requirements. This closely corresponds to the rule in RL which does not reward the agent for intermediate successes, and only rewards it when the ultimate objective is reached. Thus, instead of supplying to the agent a full goal description *a priori*, we only indicate whether the current partial trajectory is acceptable or not.

3.4.3.2 Definitions

Short of a complete mathematical formality inappropriate for the chosen level of presentation, this section will use a more precise language describing the above ideas.

Section 3.4.2 loosely introduced the *universe* as an interacting system of interconnected interacting entities.

Definition 3.1 *A vector consisting of values of all the variables that represent the state of a universe at time t is called the* state vector *or simply the* state *of that universe, and will be denoted by* $\vec{S}_t$.

Definition 3.2 *A possibly infinite sequence of states of a universe U*

$$\{\vec{S}_t\}, \quad \text{where} \quad t = 0, 1, \ldots, n \quad \text{and} \quad n \in \mathbf{N}$$

is called a trajectory *in the state space of U, and will be denoted as T.*

$T[i]$ will denote the i^{th} state in the sequence. All trajectories made of finite sequences of states will be called finite. $\#T$ will denote the length of a finite trajectory T, measured as the number of states in it.

Definition 3.3 *A trajectory* $\hat{T}$ *is called an* extension *of a finite trajectory T if* $\#\hat{T} \geq \#T$, *and where* $\forall i,\ 0 \leq i < \#T,\ \hat{T}[i] = T[i]$.

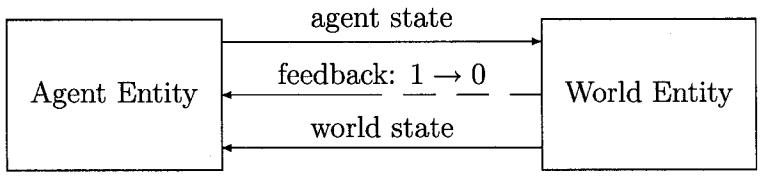

Figure 3.5: The feedback signal for binary goals

Definition 3.4 *A trajectory T is called* acceptable *for agent A, if following this trajectory results in satisfying all goal requirements for agent A.*

In other words, following an acceptable trajectory results in agent A "reaching" the specified goal.

Definition 3.5 *A finite trajectory T is said to be* conditionally acceptable *if there exists an acceptable extension $\hat{T}$ of T.*

Definition 3.6 *A set of* all *acceptable trajectories $\{T_i\}$ is called a* binary goal *for agent A, denoted by $G^2(A)$, or simply by G^2.*

In Section 3.4.4 we will generalize the notion of acceptability by allowing certain trajectories to be "more" acceptable than others.

3.4.3.3 Feedback

Within the Æip framework, goals can be communicated to agents through a special signal called a *feedback* signal. For binary goals, it will be a two-level digital signal with the value '1' indicating that the agent is currently following a conditionally acceptable trajectory. The signal will switch to '0' as soon as the system is able to recognize that the agent is no longer on an acceptable trajectory.

It is important to underscore that feedback is allowed to be at level '1' even when the agent is no longer on an acceptable trajectory. This provision allows for many real problems where we may not be able to recognize a failure right away. For example, in the game of chess, a player agent may be in a position where, objectively, it has no chance of winning, but that fact may only be apparent a few moves down the road, or only when all possible game completions have been tried from that point on. This distinction between the time when we recognize failure, and the time when that failure is unavoidable, becomes important for algorithm analysis. We, therefore, need to amend our previous statement by saying that feedback of '1' indicates to the agent only that the current finite trajectory is *not known* to be *unacceptable*, whereas a feedback of '0' indicates *with certainty* that the trajectory is unacceptable.

3.4.3.4 Examples of Binary Goals

What kinds of goals are binary? Imagine a rectangular $n \times m$ grid world where a robot, controlled by a learning agent, can move from any cell to one of the four adjacent cells. Suppose that one fixed cell always contains a "food" item and we want our robot to find the food. In the spirit of Definition 3.6, we have to separate all possible trajectories into two categories: acceptable and unacceptable. We would like to accept all trajectories that pass through the state where the robot is in the food cell, and reject all others. This is an example of a valid binary goal.

Let us consider for a moment the practical aspects of such a definition. One particularly worrisome characteristic of this goal is the fact that we are not imposing any limits on how long an acceptable trajectory should be. Indeed, according to our definition, for a 3×3 grid world, we will accept a trajectory composed of millions of states, so long as the last state is the "food" state. Even more troubling is the fact that the proposed feedback signal will never switch from '1' to '0', i.e. it will never signal a failure. After all, we cannot be sure

that a partial trajectory, no matter how long, will never pass through the food state! Since the feedback signal never changes, no learning will ever occur, and the agent is bound to eventually reach the goal by following a purely random policy. Who can argue that this is not rational behaviour?

Let us now define a better binary goal by imposing limits on the length of the trajectory. In a 3×3 grid world, we know that the shortest trajectory leading to food cannot be longer than $n+m-1 = 3+3-1 = 5$ states. We now decide to accept only the trajectories that are shorter than 6 states. At the outset of the interaction, the robot is in the starting cell and the current trajectory consists of one starting state. The feedback signal is '1', because inasmuch as we know, the current trajectory is conditionally acceptable. The signal should remain '1' until we know with absolute certainty that the trajectory has become unacceptable. In case of a grid world (as well as in many "toy" problems) we know exactly what the best solutions look like. Thus, we can switch the signal to '0' as soon as the robot stops "walking" *towards* the food cell. In learning by interaction, however, we often like to model the scenario where we don't know the solution so as to force our agents to discover it by themselves.[9] In such a case, we typically delay the notification of failure until it is trivially obvious to us that the goal will never be reached, e.g. at the end of an episode in episodic tasks. For our new binary goal, we can signal failure when the trajectory reaches length 5 and still does not contain the "food" state. It is worth noting here that a reinforcement function (in the classical RL sense) that corresponds to such a goal, is non-Markovian, and thus the Q-learning approach is expected to fail even if we adapt Q-learning to feedback signals as described below.

It is here that we must refine the notion of episodicity, as promised earlier. All binary learning tasks[10] are episodic in the sense that learning occurs as a result of failure. When

[9]In fact, in any real application we will not know the solution either, so "pretense" is a good way of testing the robustness of our algorithms.

[10]This is also true for n-ary tasks, as we will see later.

the agent moves off an acceptable trajectory, the current learning episode ends regardless of whether the environment itself is inherently episodic or not. For example, we may know with certainty, that a game of chess is lost even though we are still capable of moving our pieces. In contrast, the pole balancing task is inherently non-episodic as one can keep balancing the pole forever. Yet a learning agent will fail many times before it can keep the pole balanced, and each such failure will be considered as an end to a learning episode. To distinguish between learning episodes and episodes inherent in the task itself, we will refer to learning episodes as *trials*.

Let us pretend for a moment that we don't know the length of a shortest solution, and that we would like our robot to discover it. Can we specify an alternative binary goal that will bias the agent towards finding the shortest path? The answer is negative! The notion of acceptability is eroded by the fact that we can no longer identify unacceptable paths, and also by the fact that we now prefer shorter trajectories over longer ones. In other words, some trajectories are more "acceptable" than others. Such goals are non-binary, and they are further discussed in Section 3.4.4. It is also worth pointing out that this non-binary goal can be described by a reinforcement function that satisfies the Markovian property, and thus a Q-learning agent can find the solution. Section 3.4.4 also describes an adaptation of Q-learning to n-ary goals.

Finally, let us consider one more example of binary goals outside of the grid world domain. Suppose we are building an agent that will play a board game like tic-tac-toe or chess. Many such two-player games have three possible outcomes: a win, a loss and a tie (a stalemate, in the case of chess). All such environments are naturally episodic as the game eventually ends. We can define a binary goal whereby we accept all trajectories that end with a win or a tie position, and reject all those that end in a loss. Alternatively, we might want to reject ties as well. In other words, we have to map the three possible outcomes to the two possible feedback values of '1' and '0'. As before, the feedback signal will start at value '1' and will drop to '0' as soon as we are able to recognize that the game is lost. Note that if

our feedback signal treats wins and ties as being essentially the same, so will our agent. It might decide to tie the game even if a win is possible. Likewise, the agent will not be biased towards shorter winning games. To establish such preferences for acceptable trajectories, we must instead define n-ary goals presented later.

3.4.3.5 Q-Learning Adaptation for Binary Goals

How does the replacement of reinforcement with a feedback signal affect the algorithms that are already well-established in RL? As a proof of concept of our method we decided to retrofit the Q-learning algorithm with a feedback signal. One simple way of achieving this is to convert the feedback signal to a reinforcement signal. Since there are many reinforcement functions that will work, we chose the simplest one where the feedback of '1' always maps to a reinforcement of 0, and the feedback of '0' loosely maps to a -1 penalty. The switch of the feedback from '1' to '0' occurs only once during a learning trial, and therefore a -1 penalty will be issued only once at the end of each trial. Given this form of the reinforcement function, we see that the sum of all reinforcements that an agent can receive (or *return* in RL terminology) has only two possible values: 0 and -1. We can use the finite-horizon model for our algorithm because the learning process consists of trials, and the returns are bounded [27, p.4]. The reinforcement function just described is one of the infinite number of equivalent functions that are appropriate for binary goals, but because of its utter simplicity, we will refer to it as the *canonical reinforcement form* for binary goals.

Since the Q-learning algorithm typically initializes all the Q values to 0, we are guaranteed that throughout the lifetime of our agent each of the Q values (which are expected returns given a state and an action) will either be 0 or -1 (see analysis of the FQ-learning algorithm and Lemma 1 for details). This is a very important observation from the implementation point of view. It means that we can encode state values or state-action values (Q values) in our algorithms with a single bit! Indeed, this is where the term *binary goal* takes on a

literal interpretation. Since our conclusions were not specific to any particular binary goal, we see that the same single bit representation can be used in algorithms designed to tackle any binary goal under the assumption of interacting with a deterministic environment.

The poor scalability of tabular approaches (including Q-learning) is a well-known problem in RL. As the number of variables grows, the number of states grows exponentially, and the Q table grows quadratically in the number states! Any memory gain will allow us to tackle larger and larger problems. In a typical Q-learning implementation, we represent Q values as floating-point numbers (usually in the IEEE 754 representation), each of which occupies anywhere from 16 to 64 bits. This means that for a class of binary goals, we can reduce memory requirements of Q-learning 64-fold. As we will see in Section 3.4.4, algorithms designed to tackle non-binary goals can represent Q values with n bits, and we still manage to significantly improve on the memory requirements of our algorithms by being more precise with our representation.

We have conducted a number of experiments with this version of Q-learning, which we have referred to as Discretized Q-learning or DQ-learning, in order to experimentally confirm our conclusions. The results of these experiments are reported in Section 4.5.1.

3.4.4 N-ary Goals

3.4.4.1 Examples of N-ary Goals

In Section 3.4.3 we saw two examples of n-ary goals: finding the shortest path to food in a grid world, and preferring wins over ties in a board game. Let us consider the latter example in more detail. We would like to tell the agent entity that it should prefer winning trajectories over tying, and tying trajectories over losing. Instead of dividing all possible trajectories into two classes: acceptable and not, we want to divide them into three: winning, tying, and losing. In the most general case, we would like to divide the set of all trajectories into n classes, where n may be known in advance (in this case it is 3) or not (as is the case

with finding the shortest path to the fully "assembled" state of the Rubik's Cube puzzle). Not only do we want to divide the set of all trajectories into different subsets, but we, as experimenters, also want to order them according to our preferences. In effect, we are imposing a total order relationship on the set of all trajectories.

Thus, in a tic-tac-toe example, we can further subdivide all winning trajectories into those that win in 3 moves (shortest possible), 4 moves, or 5 moves (assuming an X player), and specify that we prefer shorter wins over longer ones. Since it is not clear to us whether we should prefer longer or shorter losing games, we can treat all losses equally. Likewise, all tying trajectories are of the same length and therefore are treated equally. Thus, the set of all trajectories can now be subdivided into 5 ordered classes.

Definition 3.7 *A function* $G : T \rightarrow [0 \ldots n-1]$ *mapping every trajectory* T_i *to a rank in the range* $[0 \ldots n-1]$ *is called a* goal of arity n *or (*n-ary goal*) for entity* A *and is denoted by* $G^n(A)$*, or simply by* G^n*.*

In the light of this definition, we can also redefine binary goals to be boolean predicates over the set of all possible trajectories. We are now in a position to generalize the feedback signal in order to accommodate non-binary goals. Instead of being initially set to '1' the feedback can start with value n, i.e. the highest rank. This will correspond to the notion of having a chance at following the most preferable trajectory from the starting point. For example, an initial feedback of 4 indicates to the tic-tac-toe player that from the starting board position it has a chance of winning in just 3 moves. After making the first three moves and still not winning, the feedback signal may drop to 3 indicating to the entity that it now has no chance of winning in 3 moves, but that it still has a shot at winning in 4. If the feedback eventually drops down to 1 it will indicate to the agent that it still has a chance of tying the game, but absolutely no chance of winning. We can designate level 0 as a total failure and a natural way of terminating the learning trial.

The choice of 4 as the highest rank and 0 as a total failure is, in fact, quite arbitrary. In many real problems we don't know how many classes of acceptable trajectories there may be. For example, in the Rubik's Cube puzzle we don't know the shortest path to the solution and, therefore, may settle on a scheme where the initial feedback signal is 0 and it is successively reduced by 1 every time the agent selects a new action. The learning trial is ended when we run out of precision with which we may be representing negative numbers. This scheme is, of course, analogous to giving a -1 penalty in RL for wasting time. So, are feedback signals really different from reinforcements? We believe so!

Our belief is partly based on the following insight. The feedback signal indicates to the agent the highest rank of a trajectory that it can hope to achieve (conditional on making all the right moves) by extending the current trajectory. The highest rank trajectories correspond to the highest returns in RL and hence the feedback signal is directly related to the value of the current state that would be obtained if the agent were to follow an optimal policy. If we also recall that signaling a failure (or partial failure, as it may be with n-ary goals) can be delayed, we see that the feedback value is rather an overestimate of the highest rank of a trajectory attainable from the current state. In traditional RL, our feedback signal is, thus, equivalent to the overestimate of the expected return from this state following an optimal policy. A drop in the feedback value is, therefore, equivalent to the immediate penalty. The key difference from RL is that the feedback signal has a clear upper bound as it never increases in value. As task specifiers, we are forced to inform the agent of the maximum expected return rather than individual rewards or penalties the sum of which may or may not be bound. The advantages of the feedback signal are further addressed in Section 3.5.

In the example of the n-ary grid world task mentioned above (finding shortest path to food), we clearly assumed that the grid dimensions are known in advance. Given the grid dimensions, we can choose a reasonable arity for the task. For a 3×3 grid, for example, we can choose arity $n = 3 + 3 - 1 = 5$, i.e. all paths longer than 4 steps will not be accepted.

What should we do, however, if the exact arity of the goal cannot be determined before the start of interaction? In other words, what should we do, if we cannot impose an upper bound on the length of the solution?

Suppose, for a moment, that we would like our agent to prove Goldbach's conjecture[11]. We are going to ignore, for the purposes of this argument, the fact that RL methods are not the best suited for the specialized domain of theorem proving. Note that in this case, we do not know the shortest path that completes the proof, or even whether such proof exists at all. This means that the feedback signal will remain at level 1 for a potentially infinitely long time, until the agent "stumbles" upon the solution. Such a feedback, clearly, does not promote learning.

We can see a number of ways of addressing such problems. First, we can assign this task the largest possible arity that our available memory capacity can allow (the larger the arity, the larger the memory requirements). If no solution is found for the goal of largest arity, we need to chose a different family of algorithms, (the current algorithm failed) or give up on the task. We can also start with a goal of small arity and gradually increase the arity if the solution is not forthcoming. This approach is similar to the Iterative Deepening A* search algorithm, where the depth of the search tree is gradually increased. Alternatively, we can define a binary goal such that trajectories with loops (i.e. the same state occurs more than once in the same trajectory) are never acceptable. The feedback signal that corresponds to such a binary goal effectively turns the agent into a depth-first searcher — when the solution is found, the path is not guaranteed to be the shortest.

[11]Goldbach's conjecture states: "every even integer greater than 2 can be written as the sum of two primes". It is one of the oldest unsolved problems in number theory.

3.4.4.2 Q-Learning Adaptation for N-ary Goals

To adapt Q-learning to n-ary goals, we need to find a translation of the feedback signal to an appropriate reinforcement function. For binary goals, we initialized all Q values to 0 and penalized the agent with -1 every time the feedback drops from '1' to '0'. For n-ary goals, we will be penalizing the agent with $(-1) \times k$ every time the feedback drops k levels. We will refer to this reinforcement function as the *canonical reinforcement form* for n-ary goals. As in the case of binary goals, the total number of different possible returns is equal to the n — the number of ranks. If the value of n is known before the interaction begins, it should be supplied to the algorithm as a parameter. This is so that the Q table can be constructed with sufficient number of bits per entry. We must also be able to indicate to the agent that the learning trial has ended, i.e. that the agent is not on an acceptable trajectory anymore. If n is known *a priori*, we can designate a 0 feedback as a failure signal. If, however, n is not known, we can start with a feedback of 0 and downgrade it by 1 every time a trajectory "changes" class. We can designate a special cutoff level of feedback (say -100) falling below which would indicate a failure. If the solution is never found with a given cutoff, we can "lower the plank", so to speak, and rerun the algorithm. A natural cutoff value is the precision limit with which we can represent negative values. Observing that the algorithm designed to work with binary goals, was just a special case of this one ($n = 2$), we can now talk of a general DQ-learning algorithm.

This general DQ-learning algorithm has been tested on the grid world domain biasing the agent towards shorter paths to the "food" cell. The shortest trajectory in an $n \times m$ grid is bound by $n+m-1$ and hence the goal is at most of arity $n+m-1$. We know, therefore, that each entry of the Q table can be represented with just $log_2(n+m-1)$ bits, i.e. 8 bits for a 100×100 grid. Observe that the same 8 bits in the floating-point representation will not be capable of accurately storing 200 integers! More extensive experimental results with n-ary goals in the domain of the LightsOut puzzle are reported in Section 4.6.4.

3.4.5 Reinforcement Philosophies

Agent actions that lead to failure are either trivially wrong (it is immediately obvious that they lead to failure) or they are determined to have been wrong after some time has passed. Assuming that the agent is dealing with a binary goal, the feedback signal should indicate failure by dropping to level 0 *as soon as* it becomes clear that the trajectory is no longer acceptable. By indicating failure upon recognition of the wrong action and thus, by aborting the learning trial, are we not "spoon feeding" the solution to our agent? In RL, it is usually considered in bad taste to give such explicit information to the agent. Instead, the learning algorithm is often advised of an overall success or failure when one or the other is instantly obvious, such as when a robot is *in* the food cell, or when the mate is obtained in a game of chess. The agent, it is believed, is supposed to determine on its own (e.g. by an appropriate backup procedure) which action was wrong in the sequence of actions that led to a failure.

If we consider the trajectory that an agent makes in the state space of the world, we can objectively identify the action which led the agent off the acceptable trajectory. When should we inform the agent that it has made a mistake? Should we do this at the end of an episode after the game is lost? But the game was already objectively lost right when that offending action was chosen! Our position is that there is no reason why we should conceal this information from the agent and keep the suspense until the end of the game.

The only reason why we might have to do this is when we ourselves don't know that the game is already lost, and have to wait until it becomes apparent to *us* as task specifiers. Imagine further, that we are dealing with a non-episodic task with potentially infinite-length trajectories. Once the trajectory stops being acceptable, it will remain unacceptable forever, and there is no objective reason to delay the notification of failure.

One can argue that such "delayed" reinforcement is necessary to see how our algorithms will behave when we ourselves do not know the solution, as must certainly be the case in any real application. This, we agree with, and we feel that a clarification is due. We believe

that in any practical application, the agent should be notified of a mistake as soon as we (the goal specifiers) can determine it. In an adaptive system, this will trigger learning, and will immediately terminate the learning trial. When we are testing the robustness of our algorithms on simulated problems, however, we will often pretend to not recognize failures until some further interaction occurs, despite our perfect knowledge of the domain. We must point out though, that such delayed notification of failure should *not* be a rule of thumb for designing reinforcement functions in general!

3.4.6 Exploratory Goals

Do there exist goals which cannot be expressed in terms of an appropriate reinforcement function? One category of such goals is clearly associated with the computability of the return. For example, although in theoretical existence, a reinforcement function that corresponds to a task of solving the halting problem cannot be specified in practice. Consequently, such goals must necessarily remain a theoretical curiosity.

As a result of our trajectory-based analysis of goals, however, we have been able to identify another category of goals which we call *exploratory*. For such goals the acceptability of a trajectory depends not only on the particular features of the trajectory, but also on how these features relate to some or all other trajectories. Hence, to pass judgment on the acceptability of a given trajectory, we must also consider other possible trajectories. Such goals present a great difficulty when we (the goal specifiers) do not have the knowledge of these other trajectories or dynamics of the entities that represent the environment.

Consider, for example, a task of finding the *second* shortest solution to the Rubik's Cube puzzle starting from some fixed state of the cube. This is clearly an example of a binary goal. What sort of return should be assigned to the first solution found? The canonical reinforcement form clearly does not work here because we are unable to determine whether

that solution trajectory is acceptable or not[12]. We could, of course, train the Q-learning algorithm, and then after convergence, make it prefer actions that result in second-best returns. Such an approach, however, is once again task specific, and would require another modification of the algorithm to now find the third shortest solution. It appears as if the only remedy in such cases is to allow for the reinforcement function to depend on the previous learning trials of an Æip agent — a sure way to confuse the latter!

We have termed these goals *exploratory* because the agent must explore the environment before we can pass any judgment on the acceptability of a given trajectory. Preliminary considerations seem to point to a solution where we augment the reinforcement or feedback signal with additional information presented to the entity so as to indicate that it deals with an exploratory goal. While we are not yet in a position to present a good way of dealing with such "nasty" goals, it is important to be aware of their existence, especially since practical applications are likely to involve poorly-known environments.

3.5 Feedback Q-learning and its Convergence

Let us assume that we are dealing with binary tasks only and that the feedback signal is provided to the agent in precisely the manner outlined in Section 3.4.3. In particular, at the beginning of the learning trial the signal starts out at the value of 1 and is guaranteed to eventually drop to the level of 0 along every non-acceptable trajectory. We are usually not interested in the trivial case in which the signal starts out at level 0 because this means that the solution obviously does not exist. For the remainder of the argument that follows, let us further assume that we are dealing with deterministic Markov environments with a finite state space, and with tasks that are of episodic nature, i.e. tasks where the goal can be reached within a finite number of steps.

[12]We assume that we don't know what the shortest solution is.

We now describe an algorithm that we provisionally call Feedback Q-Learning algorithm (or FQ-learning for short). This algorithm is an adaptation (and in a number of ways a simplification) of the general Q-learning algorithm, designed to solve precisely the binary tasks and using precisely the feedback signal described above. FQ-learning is also a special case of DQ-learning in that it deals with binary tasks only and uses a greedy action-selection policy. Rather than referring to DQ-learning in general, we would like to refer to the algorithm as FQ-learning in situations where we want to underscore the particular properties that we formally establish below. As part of our future work we eventually indend to prove the same property for the general greedy DQ-learning paradigm (i.e. N-ary goals).

3.5.1 FQ-learning Description

Figure 3.5 specifies the FQ-learning algorithm. It is described at the level of detail sufficient for the analysis that follows. As in regular Q-learning, each $Q(s, a)$ value corresponds to a state-action pair and represents the current estimate of the value (acceptability level) of choosing action a from state s. Note that in most programming languages it is easier to initialize all table elements to 0 rather than 1. Any practical implementation is, therefore, likely to flip the meaning of Q table bit values and interpret bit 0 as acceptability level 1 and bit 1 as acceptability level 0. We now show that FQ-learning, as described in the figure, is guaranteed to converge to a solution, or discover that no such solution exists. Moreover FQ-learning is guaranteed to do so in a finite number of steps.

3.5.2 Proof of Convergence

First, we show that all the Q values at any time during the learning trial are necessarily "overestimates", i.e. the value is never less than the true value of the state-action pair. In particular, this means that if the Q value is the lowest possible, then the corresponding state-action pair definitely does not belong to an acceptable trajectory. Thus, in the present

1. Allocate a Q table where each element requires a single bit of storage (recall that we are dealing with binary tasks) and initialize each bit to 1.

2. Keep conducting learning trials forever. Below is the algorithm for each learning trial:

   (a) Reset the environment to the beginning of the episode, i.e. return the Agent-Environment system to one of the several possible starting states.

   (b) While the feedback signal f is at level 1 and the episode is not over, do

      i. Observe the current state s

      ii. Greedily select and execute action $a_m = \arg\max_{a_i} Q(s, a_i)$. If several candidate actions have the same Q value, always break ties deterministically, e.g. always pick the first action in order of action indices (columns) in the table.

      iii. Observe the resulting level of the feedback signal, f

      iv. Observe the next state s'

      v. Update the $Q(s, a_m)$ table entry according to the following rules:

         - if the current value of $Q(s, a_m)$ is already 0 (i.e. action a_m is known to lead to failure from state s), leave it at 0 (no update) regardless of the observed feedback and the Q values for the next state s'.
         - otherwise if f has dropped to 0 (agent failed), set $Q(s, a_m) = 0$
         - otherwise if $\max Q(s', a)$ is 0 (i.e. all actions from next state s' are estimated to be unacceptable), also set $Q(s, a_m) = 0$
         - otherwise (both f and $\max Q(s', a)$ are 1 — acceptable), leave $Q(s, a_m)$ unchanged (also no update)

Figure 3.6: FQ-learning algorithm.

context of binary goals, a Q value of 1 is interpreted as "potentially" acceptable, but a value of 0 is interpreted as *definitely* unacceptable. Note, that in the argument that ensues, by the term *overestimate* we understand that the estimate is either greater than or *equal to* the true value, i.e. the true value is at most the value of the estimate.

The following Lemma implies for binary tasks that if the Q value is 0, the corresponding state-action pair cannot belong to an acceptable trajectory. Moreover, if $\max_i Q(s, a_i) = 0$, the state s, regardless of the chosen action, does not belong to an acceptable trajectory.

Lemma 1 *In the FQ-learning algorithm, the $Q(s, a)$ estimates in the Q table are always greater than or equal to the true values of the corresponding state-action pair.*

Proof According to the algorithm all Q values start out at level 1, which is the highest possible level. Hence, at the beginning of the algorithm's execution, the condition of the lemma holds, namely, that all Q values are overestimates. During the execution of the algorithm the only way for a Q value to become 0 is in one of two cases:

1. if the signal f drops to 0 or
2. if $\max Q(s', a)$ is 0

We show that in both of these cases, if the Q values were already overestimates *before* the update, they would remain overestimates *after* the update.

Since the level of the feedback signal is by definition an overestimate of the acceptability of the best extension of the current partial trajectory, a signal drop to 0 (as seen in Case 1) means that the corresponding state-action pair is unacceptable by definition of the feedback signal. Let us now consider the second case. Suppose that $\{Q(s', a_i)\}$ are all overestimates and $\max_i Q(s, a_i) = 0$. In the context of binary goals this effectively means that $\forall i, Q(s', a_i) = 0$. Since the true values must all be less or equal to 0 (recall, we are assuming overestimates), the s' state cannot possibly belong to an acceptable trajectory. But if the

state-action pair (s, a) led to an "unacceptable" state s', this pair itself cannot possibly belong to an acceptable trajectory because we are dealing with a deterministic and Markovian environment. Thus, if the algorithm assigns 0 to $Q(s, a)$ (as in the update of Case 2), the resulting Q value is also an overestimate. In this case the estimate and the true value are, in fact, equal.

We have, thus, shown that the "overestimate" property of Q values remains an invariant throughout the execution of the algorithm because the Q values start out as overestimates, and the updates leave this property unchanged.

□

We are now ready to turn our attention to the guarantee of convergence.

Theorem 3.1 *If an FQ-learning agent is faced with a binary task in an environment which is deterministic, Markovian, finite-state and episodic, then in a finite number of steps the algorithm will find an acceptable trajectory or determine that such trajectory does not exist.*

Proof Let us consider the trajectory followed by an FQ-learning agent during some arbitrary learning trial. Since the task is episodic in nature (must end in a finite number of steps), this trajectory is finite in length. Two possibilities exist for such a trajectory: either the Q table was not updated at all during the trial, or at least one update was made.

Let us now consider the first case, i.e. the case where no updates were made during the trial. According to the algorithm this can happen either when

1. the signal was at level 0 at the very beginning of the trial or
2. all $Q(s, a_m)$ along the trajectory are already 0 or
3. the feedback signal is at the acceptable level throughout the trial (never drops) and all $Q(s, a)$ are likewise at level 1 throughout the trial.

In the first case, no solution can possibly exist. This is a trivial and uninteresting case.

In the second case, the signal may not start out at an unacceptable level, but the agent has effectively already discovered that no solution exists. Since *all* $Q(s, a_m)$ along the trajectory are 0, let us consider the very first state s_1 on the trajectory. Since $\forall i, Q(s_1, a_i) \leq Q(s_1, a_m) = 0$ and according to the Lemma, all Q values are overestimates, it is clear that all actions out of the starting state s_1 lead to an unacceptable trajectory. Since the algorithm makes decisions deterministically, and the environment is also assumed to respond deterministically, the same exact trajectory will be retraced in all the following learning trials, with no updates to the Q values. It is clear, that in this case the algorithm will not be able to find an acceptable solution, and that it will effectively converge to an unacceptable behaviour. As the Q values are overestimates, no acceptable solution can, therefore, exist.

Finally, let us consider the third case where the feedback signal never drops throughout the learning trial and all the Q values along the trajectory are also still at an acceptable level of 1. Since the algorithm makes decisions greedily and deterministically, and the environment is also assumed to respond deterministically, the same exact trajectory will be retraced in all the following learning trials, with no updates to the Q values. It is clear, that in this case the algorithm will have found an acceptable solution and effectively converged to an acceptable behaviour.

So far, we have shown that if no updates happen during a learning trial, the algorithm converges to an acceptable behaviour (if it exists) or discovers that no such behaviour exists by converging to an unacceptable behaviour. Let us now consider a learning trial during which one or more updates to the Q values occur. Since all updates in the algorithm are "unidirectional" (the value drops from 1 to 0 and never returns back to 1) and the total number of elements in the Q table is finite (both $\{s\}$ and $\{a\}$ are finite sets), the total number of consecutive learning trials during which updates occur must, therefore, also be finite. We have, thus, shown that the FQ-learning algorithm must converge in a finite number of learning trials. Since each learning trial also consists of the finite number of time steps, we can finally conclude that FQ-learning algorithm is guaranteed to converge to a solution or find that no solution exists in a finite number of steps.

□

3.5.3 Significance of Convergence under Greedy Policy

Let us now address the significance of this theorem. It establishes one important theoretical advantage of FQ-learning over ordinary Q-learning. Specifically, the proof of convergence for Q-learning (see Section 2.4.1.6) requires that the latter visit *every* state-action pair *infinitely often*! Then, and only then, is the Q-learning algorithm guaranteed to *asymptotically* converge as the number of time steps approaches infinity. In practice, such a requirement

is virtually impossible to guarantee. This very requirement is the primary reason why an action selection strategy is crucial to the convergence of the algorithm. Instead of choosing a greedy strategy, experimenters are forced to choose ϵ-greedy (greedy only part of the time with limited random "explorations") or soft max strategies (see Section 2.4.1.6).

In contrast, our theorem allows the use of a *pure greedy strategy* and, moreover, guarantees convergence in a finite number steps, and not mere asymptotic convergence. The price we pay is that we impose restrictions on the nature of the feedback signal, the most important of which is that it must be an "overestimate" of the true acceptability of the partial trajectory. As it turns out, however, it is not that difficult to satisfy the overestimate requirement for many practical problems. The key insight here is that our feedback signal is the Interaction Learning counterpart to the admissible heuristic in A* search [55, p.76], where admissibility depends on the heuristic underestimating the remaining distance to the goal. An overestimate in the acceptability level (value of the state) overestimates how close the agent is to the solution, which is essentially the same as an underestimate in the remaining distance to the goal state. Just like A* is guaranteed to find an optimal solution if the heuristic is admissible, so is the FQ-learning agent guaranteed to find a solution (if it exists), given the overestimate nature of the feedback signal.

So how difficult is it to construct a feedback signal with the desired overestimate property? If the signal is to succeed in "driving" the agent toward acceptable behaviours, the component that emits the signal must be able to recognize what constitutes an unacceptable trajectory, i.e. a failure. Even if the signal does not indicate "failure" immediately upon entering an unacceptable trajectory, it must eventually indicate it, or no learning can possibly occur. If we are dealing with episodic tasks, in the simplest case the signal will indicate at the very end of the episode whether the exhibited behaviour was a failure, even if such a conclusion may have been apparent much earlier. So a simple form of a conforming feedback signal will be to keep the value at level 1 for the duration of the episode and to only drop it to 0 at the very end of the episode, upon failure. It should be possible to design such a signal in

virtually all scenarios when an agent is dealing with an episodic, deterministic, Markovian environment. The downside of such a simple "uninformed" signal, is that it is equivalent, in the A* domain, to a degenerate heuristic which is always equal to 0. Such a heuristic is indeed admissible (by drastically underestimating the distance to goal), but the A* algorithm in this case is reduced to a breadth-first search.

In the RL field, using overestimates as the initial Q values has long been recognized as a trick that encourages exploration. We quote below a passage from [81, p.40], where the $Q(a)$ are the action value estimates in a stateless N-armed bandit problem, while $Q^*(a)$ are the true action values:

> *Initial action values can also be used as a simple way of encouraging exploration. Suppose that instead of setting the initial action values to zero [...] we set them all to +5. Recall that the $Q^*(a)$ in this problem are selected from a normal distribution with mean 0 and variance 1. An initial estimate of +5 is thus widely optimistic. But this optimism encourages action-value methods to explore. Whichever actions are initially selected, the reward is less than the starting estimates; the learner switches to other actions, being "disappointed" with the rewards it is receiving. The result is that all actions are tried several times before the value estimates converge. The system does a fair amount of exploration even if greedy actions are selected all the time. [...] Initially, the optimistic method performs worse because it explores more, but eventually it performs better because its exploration decreases with time. We call this technique for encouraging exploration* optimistic initial values. *We regard it as a simple trick that can be quite effective for stationary problems, but it is far from being a generally useful approach for encouraging exploration.*

Our theorem shows that using optimistic initial values *in combination with* the overestimating feedback signal (i.e. FQ-learning algorithm) is not a mere trick but a legitimate

technique with a rather attractive analytical property of guaranteed finite-time convergence. Our theorem also shows that the arity-based goal analysis is not merely a framework for choosing a reinforcement strategy, or even for taking advantage of the memory savings of the DQ-learning algorithm, but that arity-based goal analysis provides a more fundamental insight into the task to be learned. Once the problem is analyzed and the arity of the task is determined, we can proceed to constructing the overestimating feedback signal that will yield the practical guarantees of finite-time convergence.

In [81, p.40] the authors go on to caution that the "optimistic initial values" technique fails in non-stationary environments. While we agree with this characterization, this is a problem for the entire family of Q-learning algorithms, including the more general temporal difference methods that rely on the Markov nature of the environment. Since a non-stationary environment is necessarily non-Markov all methods relying on the Markov property will not be successful.

We believe that the result embodied in our theorem can be further extended to general N-ary tasks (still Markovian), non-deterministic Markovian tasks (given the non-probabilistic definition of acceptability), and, potentially, to even non-episodic tasks. Our future work will attempt to address these cases.

Summary

In this chapter we presented Æip — a new general framework for setting up learning experiments. If algorithms are designed to conform to Æip, experimentation with various combinations of algorithms, environments, and interconnections can be easily set up with the help of appropriate software, such as the package we have implemented, namely, JAGUAR. Such standardization could undoubtedly speed up the research in the areas of the learning methodologies. Based on the generalizations of the classical Reinforcement Learning feedback loop that make up the Æip framework, the chapter also proposes a replacement of the reinforce-

ment signal with a new feedback signal. We argue that a feedback signal solves some of the shortcomings of traditional reinforcement functions. We also present a novel categorization of goals into *binary* versus *n-ary*, while exposing *exploratory* goals as problematic. Finally, we introduce a family of discretized algorithms (DQ-learning and FQ-learning) designed to work with the new feedback signal. Most importantly, we formally show that Feedback Q-learning has a number of advantages over ordinary Q-learning, namely, a stronger convergence result given a weaker and more practical requirement of greedy action-selection policy. The results obtained by an implementation of these concepts in JAGUAR is given in the next chapter.

Chapter 4

Solving Games with Æip

Introduction

The previous Chapter introduced Æip as a general framework for generalizing and experimenting with various learning paradigms. In this Chapter, we will show how Æip can be applied to the specific problem of learning to play games in the absence of information. To do this, we shall first highlight the principles used in the "elementary" game of tic-tac-toe, and then proceed to a more complex game called the LightsOut puzzle. In each case, in the interest of completeness, we shall describe the games and their history relevant to AI research. We will show a representation of both games within the Æip framework, as well as the details of how the Q-learning algorithm can be adapted to learn to play the games under various settings. We will also describe a set of experiments that were conducted under an ensemble of conditions and will catalogue their results. The significance of the experiments is not only in the fact that the games can be successfully learned in the setting of Æip, but also because they show that the system can be made to learn both games without ever explicitly providing any knowledge of the rules of the game to the player entities.

While tic-tac-toe is admittedly a very simple game, it has many features of the more

difficult games. Being, like chess, a two-player competitive game with three possible outcomes (win, tie, loss), it allows us to demonstrate how such a game can be modeled as an interaction between two player entities and a game entity within the setting of Æip. Furthermore, due to a limited number of moves that each player can make during each game, we can construct a number of n-ary goals which are simple to describe and understand. Finally, we also describe a novel treatment of the game where the players start out with virtually no initial information about the rules, including the basic facts that it is a two-player game where the players must alternate moves. The ultimate purpose, therefore, of presenting tic-tac-toe experiments here is to showcase the flexibility of the Æip design, to clarify the idea of goal analysis (arity), and to offer novel experiments. These concepts will then be generalized for the LightsOut puzzle.

In Chapter 3 we showed how goals can be classified based on their arity, and proposed the feedback signal as a substitute for the reinforcement signal. The advantage of using the feedback signal is that for a given goal arity there exists a feedback signal of a known shape, as opposed to an infinite variety of many different equivalent reinforcement functions. While there are still several equivalent feedback signals for a particular goal, their number is much more constrained and is, in fact, finite for an episodic Markovian task. This property of feedback is important because it eliminates many (but not all) possible errors in the manual process of converting the goal (as it is understood by the experimenter) into a signal that is directly experienced by the learning mechanism. It turns out, however, that the particular shape of the feedback signal, in combination with known arity, has another very beneficial consequence — the economy of memory representation in our learning algorithm. The experiments presented in this chapter are designed as a proof of concept for our feedback signal, as well as a demonstration of the realized economy of representation.

In particular, we will show how the benefits of such savings can be attained in practice by applying DQ-learning to a series of puzzles collectively known under a brand or generic name

of "LightsOut"[1]. While analytical methods of solving a variant of this puzzle are known, we proceed to tackle the puzzle under a different premise, namely that of putting the learning agent in a much more difficult position of having virtually no initial information about the rules of the puzzle, and also of not being able to perceive the symmetries inherent in the two-dimensional puzzle grid. In particular, without the described savings, a standard and already thrifty Q-learning solution to a 5×5 variant of the puzzle would have required at least $3\frac{1}{8}$ gigabytes of RAM with 4 bytes per table entry. Our DQ-learning based solution, in contrast, required only 5 bits per table entry, allowing us to solve the puzzle with a 1 gigabyte machine available to us at the time of this writing. Finally, we also consider a variant of this puzzle for which no analytical or algorithmic solutions are known. Our DQ-learning algorithm is equally applicable to the new puzzle and can find solutions to such a puzzle with exactly the same amount of effort, while still not knowing the rules of this puzzle at the outset.

4.1 Playing Games — A Survey

AI research has always been interested in building algorithms that play various games. It is a widely held belief that successfully playing a challenging game, such as chess, requires precisely the mental capabilities that we normally call "intelligence". We can identify two distinct but complimentary approaches to building such game playing algorithms. The *search* approach involves searching the game tree to select the best move, usually employing a heuristic evaluation function of the game states. The *minimax* algorithm [55, p.112] exemplifies this approach suitably. The *learning* approach, on the other hand, attempts to minimize the search by learning a good heuristic evaluation function. In RL, for example, we learn a value function that assigns the expected sum of future rewards (a value) to a game state.

[1]The history of the puzzle is described in Section 4.6.1.

The value function is, thus, equivalent to the heuristic evaluation function.

We believe that combining these two approaches is not only possible but rather promising, whereby a poor heuristic can be improved through learning. This would, in turn, reduce the look-ahead search and, analogously, a look-ahead search would aid in learning the heuristic function. Such mixed or hybrid approaches have yielded a number of success stories, such as the famous checker's program by Samuel [74], and the subsequent TD-Gammon program by Tesauro [83]. Unfortunately, other attempts to use the same approach for other games, such as chess, have not been so successful. The problem seems to lie in the non-convergence of the underlying learning algorithms. We believe, however, that such mixed approaches have tremendous potential and thus deserve further exploration.

The search approach requires complete knowledge of the game *model*, i.e. the rules of the game being played. In particular, we must know what moves are available from the current state, what the next state would be for a given move, and what states are *terminal*, i.e. winning, losing or tying. This is fundamental to the search approach because, after all, we would not be able to expand the game tree if the model was not known. The tough predicament of not knowing the game model is usually not a problem for the learning approach, though. In fact, the RL literature abounds with the so called *model-free* techniques such as Q-learning, where the algorithms never store any explicit information as to which state would follow if a given action is selected. Yet, even in these model-free algorithms, the system eagerly attempts to exploit certain aspects of the game model. For example, not all moves are valid in all states of the game, and so we typically limit the attention of our RL algorithms to moves that are valid in the current state. Likewise, the fact that in a two-player game the opponents alternate moves is normally taken advantage of by only allowing the agent to select actions when it is, indeed, its turn to make a move.

In this chapter we will report on a series of learning experiments where agents learn to play games with as little initial knowledge of the game as possible. In particular, we will employ a variant of Q-learning to learn to recognize which moves are valid for a given game

state, and whether it is currently the agent's turn to make a move. At the same time the agent will, of course, attempt to learn to not lose to the opponent, which is feasible in tic-tac-toe, because winning is only possible if the opponent plays suboptimally. As we stated earlier, we start with a game of tic-tac-toe because despite its simplicity, it is a sufficiently interesting game which possesses many features of more difficult games. Being, like chess, a two-player competitive game with three possible outcomes (win, tie, loss), it allows us to demonstrate how such a game can be modeled as an interaction between two player entities and a game entity within the setting of Æip. Furthermore, due to a limited number of moves that each player can make during each game, we can construct, for our DQ-learning algorithm, a number of n-ary goals which are simple to describe and understand. Finally, we also describe a novel treatment of the game where the players start out with virtually no initial information about the rules, including the basic fact that it is a two-player game where the players must alternate moves. As mentioned earlier, the goal we seek in presenting tic-tac-toe experiments here is to showcase the flexibility of the Æip design, to clarify the idea of goal analysis (arity), and to offer novel experiments. These concepts will then be generalized for the LightsOut puzzle.

4.2 The Game of Tic-tac-toe

Tic-tac-toe (known in England as *Naughts and Crosses*) is a popular children's two-player board game with simple rules. The game board is a square subdivided into 9 cells by a 3×3 grid. Before the game begins, the players agree which one is going to play naughts; the other player plays crosses. By convention, the crosses player always makes the first move, which may be a significant advantage if the opponent (naughts) is playing suboptimally. The players take turns by drawing (or placing) their corresponding figure (a cross or a naught) into an empty cell. Initially there are 9 empty cells and so the crosses player has 9 possible openings. The opponent then has only 8 empty cells left to choose from. The objective of

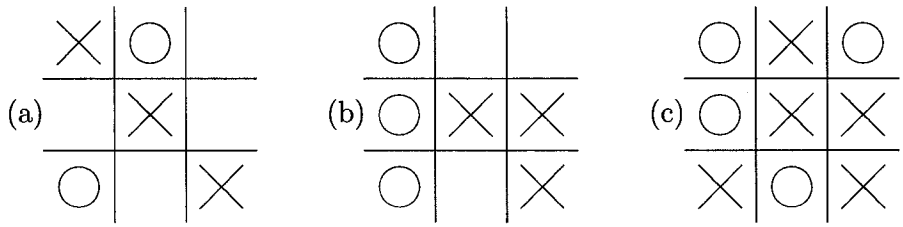

Figure 4.1: Three possible game outcomes: (a) crosses win, (b) naughts win, (c) a tie

the game is for a player to place three of his own figures in a row - horizontally, vertically or diagonally. The first player to achieve this wins the game. If none of the players can win and all the cells have been occupied, the game ends in a tie. Figure 4.1 shows these three possibilities. If both players follow the optimal strategy, the best outcome that each can hope for, is a tie.

Tic-tac-toe has been a popular "toy" problem for game playing research since the early days of AI. Michie [46] constructed a mechanical system called MENACE (Matchbox Educable Naughts and Crosses Engine) which consisted of a matchbox for each possible game position containing a number of coloured beads where each colour represented a different move. This was an early example of a trial-and-error RL system that significantly influenced future research. Nilsson used tic-tac-toe for his example of the minimax and alpha-beta procedures in a pure search approach [55, p.115]. Like many other AI researchers, we have opted to choose the tic-tac-toe domain, as an initial *prima facie* example, for a number of reasons, not the least of which are its inherent simplicity and the resultant ability to manually verify the decisions of the algorithm.

What could be the motivation for us to train agents to play a game without providing the underlying rules? We were guided by three primary interests. First of all, as explained in Chapter 3, we are interested in the generalization of the agent-environment feedback framework employed in RL. When writing general-purpose learning algorithms for such

a framework, it is the norm for the agent to assume very little about the environment with which it is interacting. Otherwise the algorithm is simply no longer general. As a consequence of this model of computation, our tic-tac-toe player entities are not permitted to take advantage of the specifics of the game beyond the rudimentary information already afforded by the framework. Our second motivator was our interest in games and game playing algorithms in general. Tic-tac-toe, being a well-understood, simple and yet non-trivial, two-player game, lends itself easily to such a study. Lastly, the learning paradigm and its limitations are of great interest to us. The fact that a simple Q-learning agent can learn the set of valid moves in a given state and the alternation of moves between opponents, is a powerful affirmation of the learning paradigm, in general, and of RL, in particular. It is also fascinating to see how the lack of knowledge affects the convergence and the quality of learning.

4.3 Æip Specification of Tic-tac-toe

4.3.1 Entity diagram of the Game

In accordance with Æip, we must first decide how many entities would be required to experiment with a game of tic-tac-toe. Since it is a two-player game, we will need 3 entities as modeled in Figure 3.2. Two of these entities will represent the two players, and the third will represent the tic-tac-toe board, which also encodes the rules of the game. In all our experiments, at least one player entity will be a learning agent. The other could also be a learning agent, but might also be an optimal player, or an interface to a human player.

Next we specify the inports and outports for our entities. The tic-tac-toe world presents its board state as an array of nine outports - one for each square on the board. The values for each outport will be numeric and limited to three values, namely, 0 (empty), 1 (cross), and 2 (naught). While this arrangement is not intended to be optimized so as to facilitate

learning, it is a simple and straightforward representation. Clearly, alternate arrangements are also conceivable.

The agent entities will have an observable state that consists of a single value, which indicates the square where the player wishes to place its piece. Since there are nine squares, the values will range from 0 to 8. The reinforcement signals for the player will contribute an additional inport per agent, and one composite outport to the world. Each agent entity will, therefore, have a total of 10 inports and one outport, while the world will have 10 outports and two inports. The corresponding entity diagram is shown in Figure 4.2. Note that the composite reinforcement consisting of an array of two values is split by a special *array-splitter* entity.

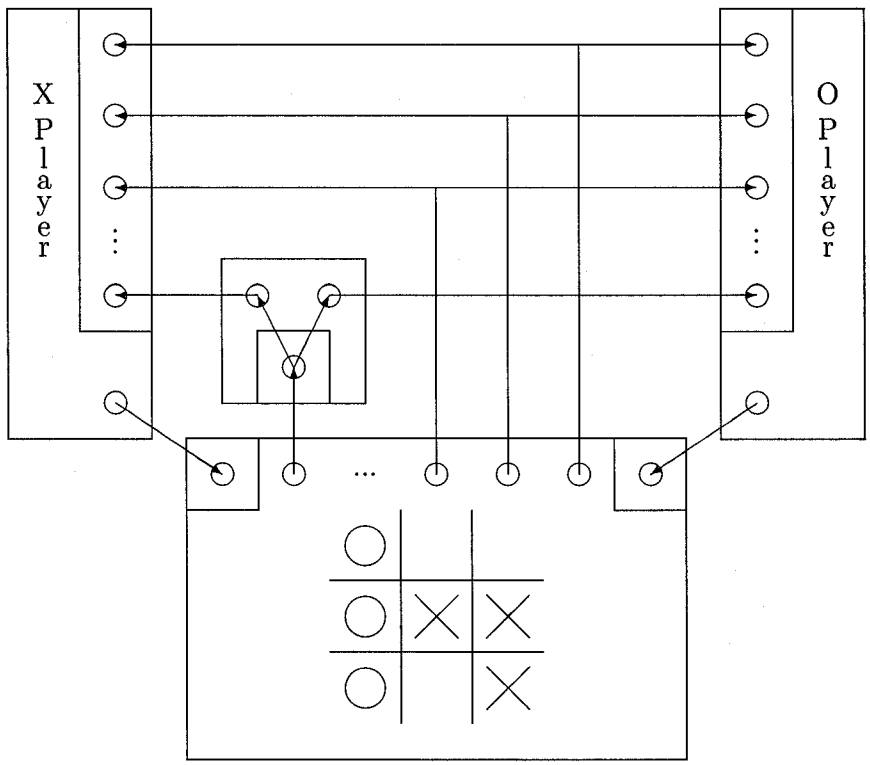

Figure 4.2: Tic-tac-toe entity diagram

4.3.2 Scheduling the Interaction of Entities

In addition to the connection topology outlined above, we now decide how the interaction between the entities will proceed. The question we address is whether we can simultaneously update all three entities involved in the game. While this is conceptually possible, this approach presents a serious difficulty when it concerns the reinforcement itself. For example, before a learning trial begins, the tic-tac-toe board is empty, and both agents have selected certain actions. Thus, it could be that both wish to place their piece into the top left square — square 0. On the first tick of a universal clock, all three entities update their state. The world places a cross into square 0, chooses two reinforcements for each of the agents, and changes the corresponding outport values. Each of the two agents select another action by changing the value on their outports. The reinforcements issued by the world after the first tick will not be processed by the agents until the next tick, by which time a second action will have been chosen. In other words, there is a one time-step delay between the time the action was selected and the time the reinforcement is processed. Figure 4.3 clarifies this by showing the evolution of the world (consisting of a state s_t and a reinforcement r_t) and one of the agents (with state a_t). The reinforcement r_2 given for action a_1 chosen in tick 1 is processed by the agent only during tick 3.

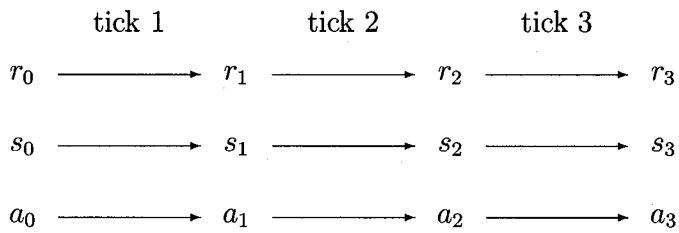

Figure 4.3: Simultaneous scheduling of all entities

The Q-learning algorithm used in the experiments, on the other hand, needs to update the Q table right after the reinforcement is issued. We can choose two alternative ways of

resolving this problem. One approach is to add a variable to the Q-learning algorithm that remembers the action selected two steps ago. Alternatively, we can disallow simultaneous updates of the world and the agents. The first approach is not too difficult to implement, especially since some versions of Q-learning already record the complete interaction history for the whole episode, and perform the updates in a batch mode at the end of the episode. We have chosen, however, the second alternative, in part, so that we can test the flexibility built into the Æip design. We have opted to alternate between the updates of the world and the simultaneous update of the agents. In other words, all three entities are updated only on every other tick of the universal clock alternating between the agents and the world. Both agents update themselves simultaneously on the first tick while the world is dormant. On the second tick, the world updates itself while both agents are waiting, and so on. The resultant interaction is depicted in Figure 4.4.

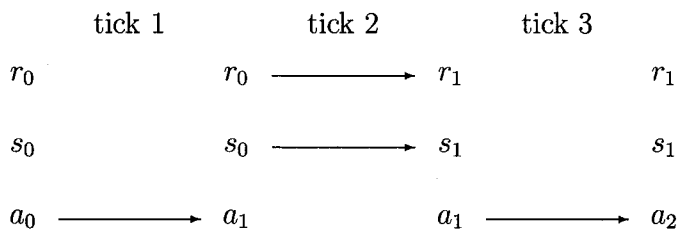

Figure 4.4: Scheduling for our experiments

The difference between our current model and that, which would have been normally done in RL, is that both agents are updated simultaneously. Such an updating is, indeed, required if the agents are to learn the *concept* of the alternation of moves. Note that if both agents choose an invalid move, the tic-tac-toe world ignores both of them and does not change the board until a legal move is made by the agent entity whose turn it was at the time.

4.3.3 Reinforcement Signal

Now that we have described how the interaction is to proceed, we address the question of the necessary reinforcement signal itself. A classical reinforcement function for a tic-tac-toe game assigns a reward value of +1 to the action that leads to a winning state, the reward of 0 to one that leads to a tie and, finally, a penalty value of -1 to the action that entails a loss. Since the total number of moves in a tic-tac-toe game is rather small, no preference is, thus, imputed to shorter winning games.

Can this reinforcement function be adapted to the case when the agents do not know which moves are legal and which are not? The answer turns out to be negative, because if we do not penalize the agent for wasting time, it might get stuck always selecting an illegal move. To make sure that we are not explicitly telling the agents which moves are legal, we chose a simple reinforcement schedule, which assigns a penalty value of -1 to every action that does not end the game, regardless of whether the move was legal or not. The winning actions were reinforced with a +10 reward and the losing actions were discouraged with a -10 penalty. Ties resulted in both agent entities receiving a reward of +5. The absolute values of these reinforcements are quite arbitrary as they only affect the "interesting" range of *temperature* values as described in the following section.

If we want to make the player aware of the illegal moves, we should modify the reinforcement function so as to explicitly penalize the illegal moves and remain neutral towards the legal moves that do not end the game. Making an illegal move can be considered just as bad as (if not worse than) loosing the game, and consequently, penalizing an illegal move by the same value of -10 is only natural. All other non-game-ending moves receive a reinforcement of 0. Both of these reinforcement functions were used in our experiments.

4.4 Featured Experiments with Q-learning

Among the many experiments conducted within the JAGUAR framework, we underscore three sets of results. We trained our agents to play tic-tac-toe by providing them with differing levels of explicit information about the rules of the game. In the first set of results, we provide the agent with as little information as possible. The agents start out not knowing which moves are legal, whether players must alternate moves and what might constitute a winning or losing position. In the second experiment, the agents were made "aware" of the alternation of moves by virtue of the interaction schedule which did not require an agent to select a move out of turn. In the final experiment, we provide the agents information about both alternation of moves *and* the immediate strong penalty for illegal moves.

4.4.1 Learning Algorithm

In all three experiments described below, we used an implementation of the Q-learning algorithm with slow update as outlined in Section 2.4.1.6. In our JAGUAR implementation, the dimensions of the Q table were computed from the ranges of values appearing on the individual imports of the agent. Since all our experiments are episodic, we set γ to unity, i.e. no discounting of the rewards was done. Both players were learning to play at the same time and, therefore, we utilized a non-deterministic version of Q-learning to ensure convergence. While the method of gradually reducing the learning rate is not specific to any domain, our experiments have shown that in a tic-tac-toe domain, setting the learning rate to a constant value is sufficient, and often leads to faster convergence. Both agents used a constant learning rate $\alpha = 0.5$.

We have chosen for our experiments the soft max approach as described earlier in Section 2.4.1.6. From a practical point of view, we have observed that it was sufficient to set $\tau \geq 1$ to get an action selection strategy that is quite close to random, and, thus, to encourage exploration for the kinds of Q values used in our experiments with tic-tac-toe. In fact,

in the experiments reported here, we used only two values for the temperature: 0.5 — to encourage moderate exploration, and 0.1 — to occasionally switch to a greedy policy. In our setting, where both opponents are learning at the same time on mutually played games, a higher value for the temperature will improve the quality of the game by forcing the agents to explore new combinations. If, on the other hand, the agents always followed a greedy policy during training, the game quality would suffer because the agents would have both converged to mutually mediocre policies. In other words, the agents would have learned to play well against each other, but not necessarily against a tough opponent. Since the soft max approach requires a random action selection, we used a standard pseudo-random generator[2] seeded with value 232323 for the crosses player, and value 565656 for the naughts player. To ensure repeatability, the same seeds were used in all three experiments.

4.4.2 Experiment 1: No Information about Rules

In the results presented here, two Q-learning agents were playing against each other, both starting with virtually no knowledge of the game of tic-tac-toe.

30,000 games were simulated in 300 batches of 100 games per batch. The agents selected their actions simultaneously, and only one of the two actions affected the board according to the "alternation of turns" policy known by the tic-tac-toe world, but not known by the agents. The latter were also allowed to select invalid moves and were supposed to learn which moves were not valid. We set the overall temperature $\tau = 0.5$ to encourage exploration, and changed it to 0.1 (greedy policy) for every 10^{th} batch in order to verify whether the learning process had converged. Figure 4.5 shows a plot of the running averages (computed over each batch) for the percentages of the three game outcomes in relation to the total number of games played.

According to the plot, even after having played 30,000 games, the agents still do not seem

[2]Implemented by `java.util.Random` class in version 1.3 of JDK.

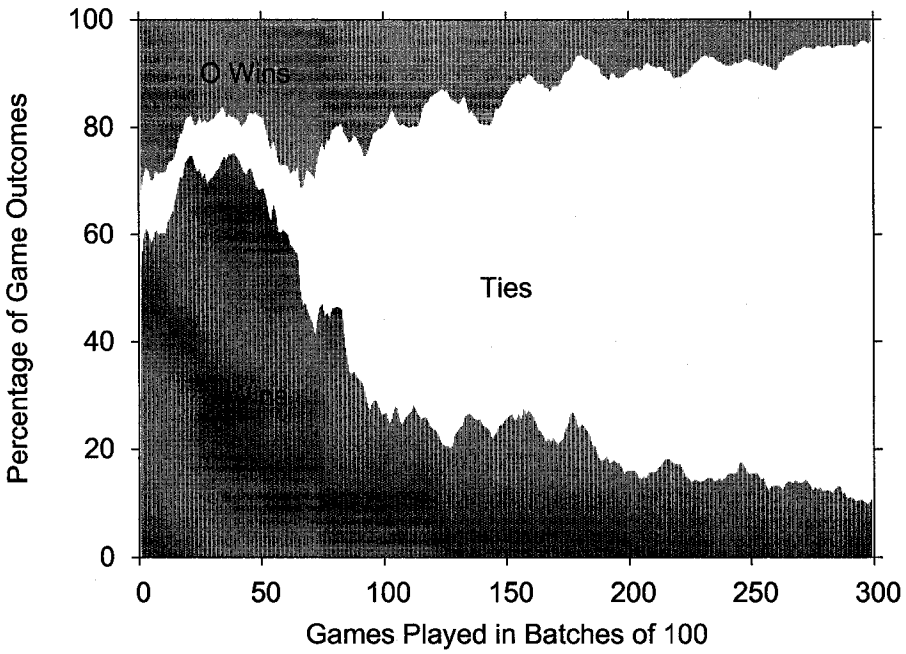

Figure 4.5: Percentage of game outcomes as more games of tic-tac-toe are played. The plot shows only the batches with τ set to 0.5. The number of games is shown in hundreds.

to play optimally; games are still being lost. The reason is that a temperature of 0.5 introduces a certain degree of randomness into the action selection process. Even when Q-learning has already converged, suboptimal actions continue to be selected. Figure 4.6 demonstrates this phenomenon by plotting only those "greedy" batches with $\tau = 0.1$. Observe that the algorithm has practically converged after 10,000 games.

The first experiment clearly demonstrates that it is possible to learn to play the game with no prior knowledge of the fact that players must alternate moves, let alone the knowledge of which moves are legal or which ones make up a winning strategy.

The best indication of learning the alternation of moves would be the agent electing not to make any moves when the opponent has the turn. We did this by allowing the agents to make a regular move during the opponent's turn. After sufficient training, the agents realized that

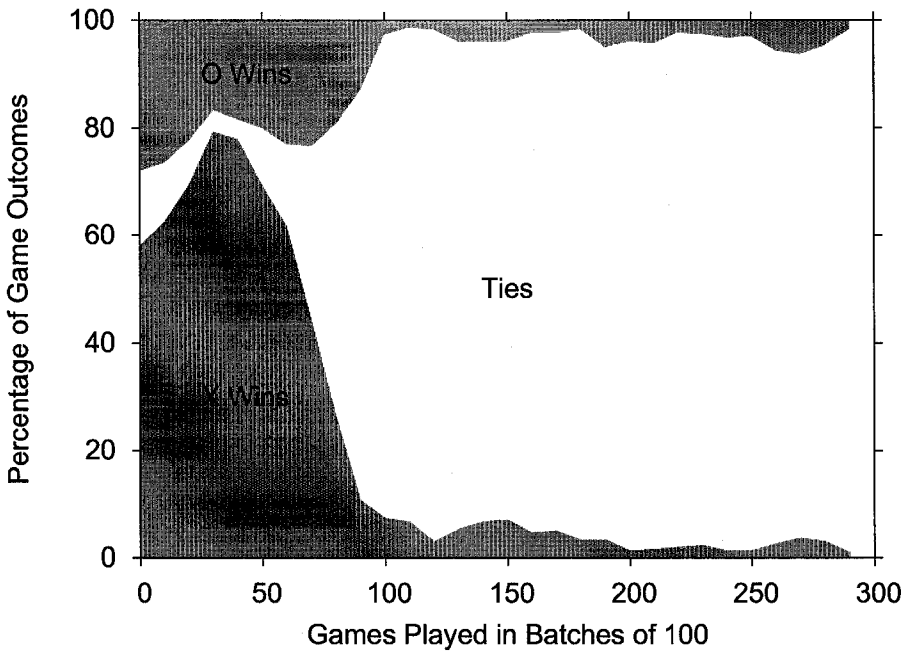

Figure 4.6: Plot showing only "greedy" batches of tic-tac-toe with τ set to 0.1. In this setting, as in Figure 4.5, the players have no knowledge of alternation of moves or any other rules of the game, for that matter.

in certain states of the tic-tac-toe world, all actions have equal values associated with them and there is no reason to prefer one over the others. This is seen as the agent's "indifference" towards making a decision at the time when the opponent has the turn.

4.4.3 Experiment 2: Alternation of moves is "known" but not legality

By withholding the information about the alternation of moves from our agents we expect them to spend more time learning to play the game. The second experiment demonstrates how much faster the agents learn to play when they are only allowed to make a move on their

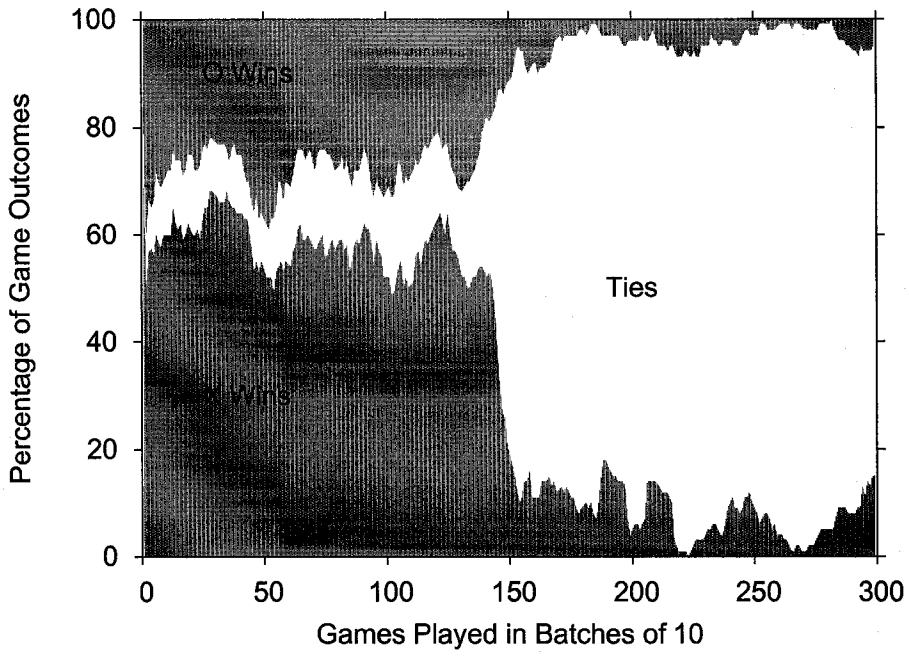

Figure 4.7: Learning of tic-tac-toe when alternation of moves is known to the agents. The number of games is shown in *tens*.

own turn. In this experiment the entities were scheduled in the usual tic-tac-toe fashion:

$$\text{X} \rightarrow \text{World} \rightarrow \text{O} \rightarrow \text{World} \rightarrow \text{X} \rightarrow \text{World} \rightarrow \ldots$$

although the agents could make illegal moves. Figure 4.7 shows the result of playing 3,000 games in the same 300 batches of *only 10 games* per batch. As before, the temperature was switched from 0.5 to 0.1 for every 10^{th} batch. We observe that the agents achieve respectable results after only about 1,500 games showing a nearly seven-fold speedup in convergence, when compared to the experiment where alternation of moves is not known to the agents.

Figure 4.8 presents a plot of only the greedy batches where we observe the same seven-fold effect except, we see complete convergence to 100% of playing ties.

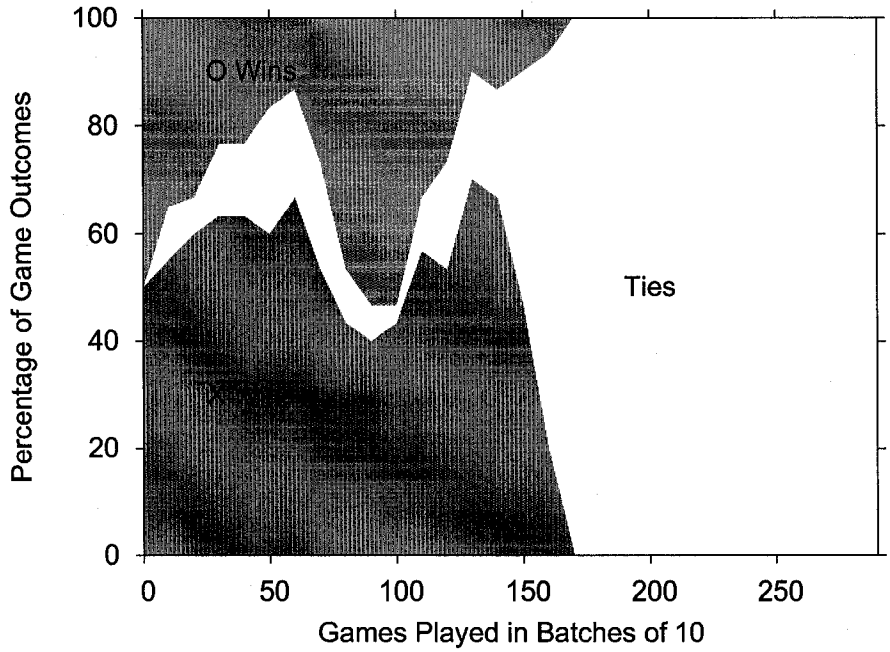

Figure 4.8: Learning with alternation of moves. The plot shows greedy batches only.

4.4.4 Experiment 3: Move alternation and legality are "known"

Let us now confirm, that by additionally providing the information about which moves are illegal, we speed up the convergence. For this experiment we utilized a different reinforcement function. As described in Section 4.3.3, it penalized the illegal moves with a penalty of -10, and issued a "neutral" reinforcement of 0 for all other moves that did not lead to the termination of the game. The game ending moves were reinforced as before with +10 for a win, +5 for a tie, and -10 for a loss.

We must point out that there exist other methods according to which a Q-learning agent can be said to "know" which moves are illegal. For example, instead of initializing the Q table with all zeros, the Q values that correspond to illegal moves could have been pre-initialized to some strong negative values (e.g. the same -10) to discourage their selection

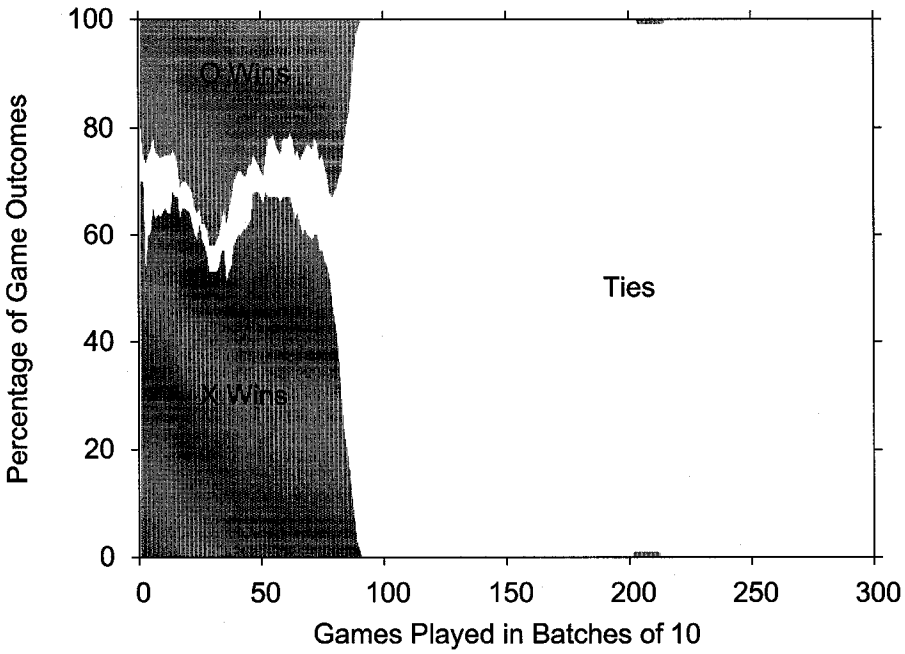

Figure 4.9: Learning tic-tac-toe with alternation and legality of moves known. The number of games is shown in tens.

by the soft max algorithm. In fact, this method may be considered "better" in the sense that the agent "knows" right from the start which moves to avoid without any need for learning. In contrast, our agents in this experiment had to first try an illegal move and only then receive a strong discouragement in the form of penalty of strength -10. The Q table, however, is then immediately updated having the same effect as pre-initializing it to -10. This approach, of course, has the drawback of forcing the agent to try the illegal move once, but it also has an advantage that tipped the scale in its favour. Pre-initialization, in effect, makes the agent (and the underlying algorithm) domain-specific, whereas the Æip philosophy is to build general purpose entities in order to be able to reuse the same agent code in various environments.

Information about alternation of moves was also implicitly provided as in the previous

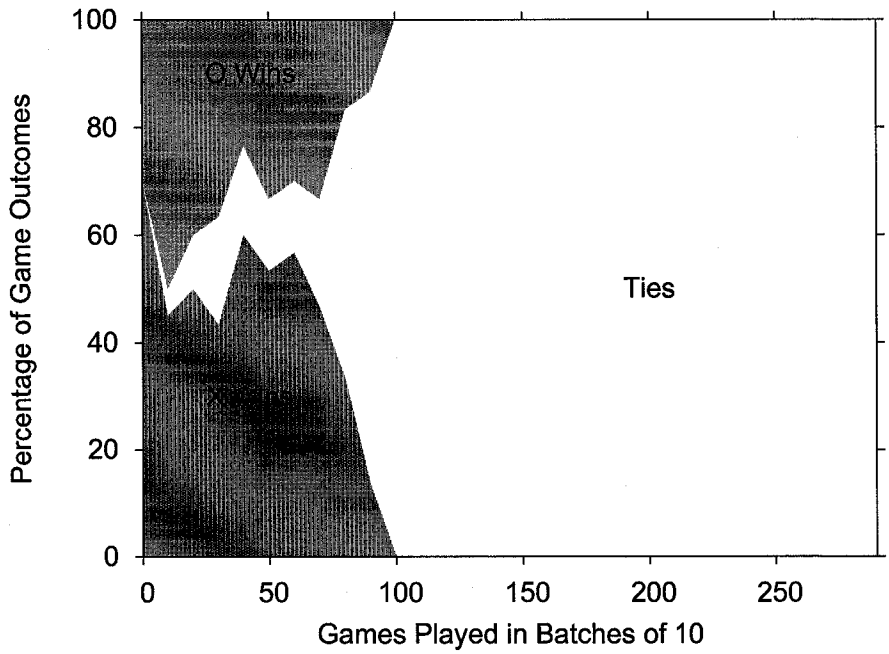

Figure 4.10: Learning tic-tac-toe with alternation and legality of moves known. The plot shows greedy batches only.

experiment.

Figure 4.9 shows the result of playing 3,000 games, again in 300 batches of 10 games per batch. As before, the temperature was switched from 0.5 to 0.1 for every 10^{th} batch. We observe that the agents achieved virtually complete convergence after a mere 800 games, even with the exploratory temperature of 0.5. The sudden "hiccup" starting at batch 202 is due to a single victory by crosses and another victory by naughts. It can be explained by the inherent randomness of action selection and the relatively high temperature.

Figure 4.10 presents a plot of only the greedy batches where we observe the same rapid convergence for batch 80, except we do not expect to see any "hiccups" as more and more games are played.

We must point out here that by convergence in all three experiments, we mean that the

two player entities have managed to learn *each other's* behaviour so well, that their games at the end always result in ties. We are not suggesting that both agents have learned the best strategy of play in tic-tac-toe; rather we believe that they have learned not to lose to one specific opponent. To ensure that the players learn the optimal strategy against *any* opponent, they must be individually trained by playing a "perfect" player. Such experiments are also possible and were in fact conducted in JAGUAR. We decided to omit the results as the novelty of the approach has already been demonstrated in the experiments presented here. Rather, it would be more beneficial to move ahead to the concept of using binary goals and then to the LightsOut puzzle.

4.4.4.1 Concluding Remarks

In conclusion, we have used the Æip framework to experiment with learning a two-player game of tic-tac-toe. We have shown one possible representation of the game in terms of various entities and their interconnections involved, underscoring again the generality and flexibility of the framework. One of the major design goals of Æip is to build general-purpose easily interchangeable entities. In this way, the same configurable learning agent can be reused in many different experiments. We, therefore, wanted to build our Q-learning agent with no domain-specific information about the game. As a result, we naturally focused our attention on a particularly interesting set of experiments that demonstrate how an agent can learn to play a new game without any prior knowledge of the rules of the game, let alone a good playing strategy. The experiments clearly show that even a model-free method such as Q-learning manages to learn the game under such conditions. The price one pays for this lack of information is addition computation resulting in over 12 times slower convergence.

4.5 Feedback — Æip Alternative to Reinforcement

We showed in Chapter 3 how goals can be classified based on their arity, and proposed the feedback signal as a substitute for the reinforcement signal. The advantage of using the feedback signal is that for a given goal arity there exists a feedback signal of a known shape, as opposed to an infinite variety of many different equivalent reinforcement functions. While there are still several equivalent feedback signals for a particular goal, their number is much more constrained and is, in fact, finite for an episodic Markovian task. This property of feedback is important because it eliminates many (but not all) possible errors in the manual process of converting the goal (as it is understood by the experimenter) into a signal that is directly experienced by the learning mechanism. It turns out, however, that the particular shape of the feedback signal, in combination with known arity, has another very beneficial consequence — the economy of memory representation in our learning algorithm.

The experiments presented in this chapter are designed as a proof of concept for our feedback signal as well as a demonstration of the realized economy of representation. The following sections use our DQ-learning algorithm which is an adaptation of Q-learning described earlier in Chapter 3. In DQ-learning, the reinforcement signal is expected to be of canonical form, and, as a result, the individual Q values are stored in bit fields of fixed length, starting with a single bit and larger.

4.5.1 DQ-learning with Binary Goals

We decided to apply DQ-learning to our domain of tic-tac-toe without optimizing the algorithm for this domain specifically[3]. We used the same JAGUAR implementation of Æip, and exactly the same entity diagram shown in Section 4.3, except that only one player entity was connected. Our implementation of the tic-tac-toe world entity was such that when only a

[3]We shall later show how this same philosophy can be used for other application domains.

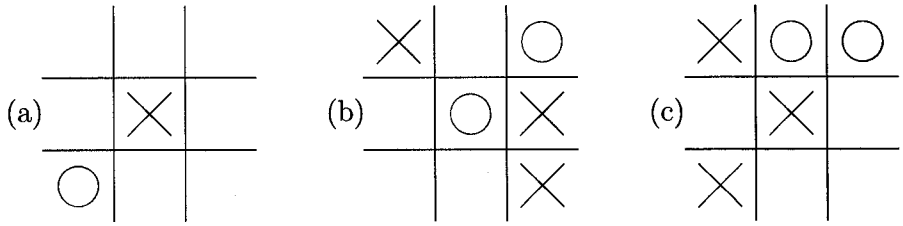

Figure 4.11: Tic-tac-toe heuristic: (a) $H_X = 1$, (b) $H_X = -1$, (c) $H_X = 4$

single player was connected, it emulated the other player by following a heuristic strategy[4].

The heuristic value $H(p)$ of a position p is the difference between the *strengths* $S_i(p)$ of the two players. Specifically,

$$H_X(p) = S_X(p) - S_O(p) \quad \text{and} \quad H_O(p) = S_O(p) - S_X(p) \quad \text{or} \quad H_O(p) = -H_X(p).$$

The strength of a given player is calculated as the sum of the player's pieces on all rows (vertical, horizontal or diagonal) that can still be completed. Figure 4.11 shows examples of the heuristic values for the crosses player assigned to three different game states. In position (a), the crosses player has three possible rows open, each contributing 1 to the total strength of $S_X = 3$, while the naughts player has only 2 rows open, for a total strength $S_O = 2$. In position (b), the naughts player has the same number of open rows as the crosses player (namely, two), but it has a stronger position, because it has a nearly completed row along the diagonal, with a strength $S_O = 1 + 2 = 3$. In contrast, position (c) shows a much stronger crosses side with naughts having only a single open (rightmost vertical) row. As can be easily seen, position (c) leads to an inevitable victory by the crosses side.

The action selection by the heuristic player was done in a "greedy" and deterministic fashion, where the first action with the best immediate prospects was selected based on the heuristic evaluation of the resultant board positions. The greedy approach was chosen

[4]It can also be configured to follow an *optimal* strategy by exhaustively searching the game tree.

specifically to eliminate non-determinism in the environment. We shall presently show in Section 4.5.2 how the DQ-learning can be adapted to deal with non-determinism. To enable greedy selection, we needed to specify a temperature of 0 for the emulated player. This temperature value is interpreted by JAGUAR in a fashion that is qualitatively different from any other temperature no matter how close to zero. A temperature of 0.1, for example, could also be said to be greedy, except that the softmax approach will still assign an equal selection probability to all "equally" best actions, thus introducing a source of non-determinism. In JAGUAR, a temperature of exactly 0 disables softmax in favour of a deterministic greedy approach, which always selects the same action (usually the first one that it discovers) among several best choices.

The learning player entity received the canonical reinforcement for the binary goal of playing tic-tac-toe: all illegal moves and moves leading to a loss were given an immediate penalty of -1, whereas all other moves were given a "success" reinforcement of 0. Thus, in accordance with our definition of the feedback signal, making an invalid move into an occupied square resulted in an immediate failure and the termination of a trial. As discussed in Section 3.4.3.5, a combination of the deterministic environment and the canonical reinforcement form leads to single bit representations for each entry in the Q table. Initially, all entries were initialized to 0 and switched to -1 upon receiving a -1 reinforcement (i.e. failing). To actually store the two possible values in a single bit, the -1 value was encoded as a bit with value 1. We must underscore that, due to the determinism of the environment, the transition from value 0 to value 1 for a given entry would occur only once, if at all. In other words, once the value converged, it never had to be revised. This corresponds directly to the convergence of the classical Q-learning for a deterministic MDP environment. In Section 4.5.2 we shall present an example of dealing with a binary goal in a non-deterministic MDP environment.

Since we did not want to use an algorithm custom-built for this domain[5], our Q table consisted of *states* × *actions* = $3^9 \times 9 = 177,147$ elements. If in a Java implementation, the single precision floating-point numbers (32 bits per value) were to be used, the resultant Q table would occupy $177,147 \times 4 = 708,588$ bytes, or more than 2/3 of a megabyte. In fact, since Java encourages the use of double precision floating-point values, it would not at all be unreasonable that an average implementation (including our own) would be using double precision values that require 64 bits of storage each. That doubles the size of the table to 1.35 megabytes. By contrast, the same table using the DQ-learning mode took only $177,147 \times 1/8 = 22,144$ bytes, or just under 22k! This reduction in memory was achieved without taking advantage of the specifics of our domain, but rather by a general observation that a binary goal is sufficient to describe the task of playing tic-tac-toe[6]. As expected, the algorithm's performance, in general, and convergence, specifically, were not affected by our modifications.

We have conducted two experiments using DQ-learning to play against a deterministic opponent. In one experiment, we learned to play against the emulated crosses opponent, and in the other — against the naughts opponent. Each experiment required us to configure the algorithms with the following information: the number of ports, the expected range of values on each of them, and the temperature to be used for action selection. As in other tic-tac-toe experiments (e.g. in Section 4.3) a learning player entity will have 10 inports (9 for each cell of the game board and 1 for reinforcement), and 1 outport with the proposed move. Each of the 9 cell inports has 3 possible values: 0 (empty), 1 (cross), and 2 (naught), while the move outport has a range of [0..8] — one for each board cell.

Based on the way we initialized the Q table (i.e. with all zeros, meaning that each state-action pair is initially believed to lead to the goal) and the fact that we use the canonical

[5]We did not want to take advantage of inherent symmetries of tic-tac-toe, for example.

[6]With the caveat, of course, that we do not distinguish between the winning and tying game outcomes.

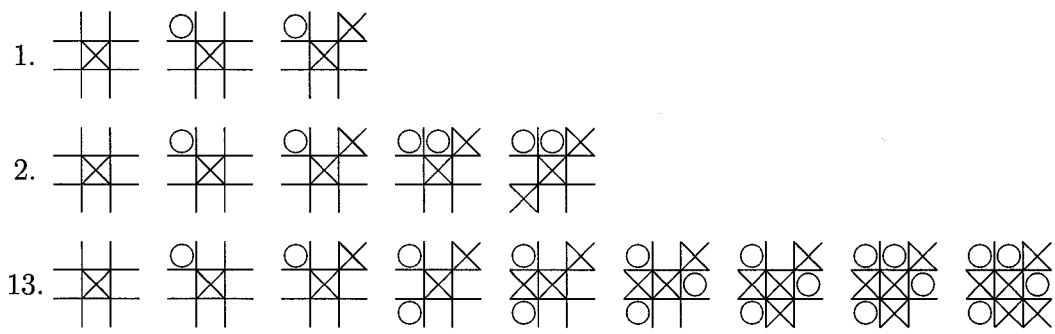

Figure 4.12: Trials 1, 2 and the first successful game 13 against the crosses.

reinforcement form for binary goals, we realized that it is sufficient to always use a greedy (and even deterministic) action selection strategy. The random nature of the softmax approach was necessary to drive exploration in the hopes of avoiding a premature conversion to a locally optimal policy. With binary goals there are no local optima as all successful trajectories are deemed equally acceptable. We, therefore, decided to switch to a deterministic greedy action selection strategy by setting the DQ-learning temperature to exactly 0.

4.5.1.1 Playing against the Crosses

Since both the tic-tac-toe world and the learning player entities used the deterministic greedy action selection, the learning trials naturally degenerated to always playing the *same* game, with the subject player gradually learning not to lose. Not surprisingly, it took only 12 trial games before the naughts side had completely converged. Most of the trials ended in neither a win, a tie, or a loss, but in a failure to make a legal move by the learning player. In the remaining trials the naughts player is learning which of the legal moves lead to a loss and should be avoided. Figure 4.12 shows the resulting tie game played as well as two of the trials leading to it. The first trial terminated after the naughts player proposed an illegal move into the top left square already occupied by a naught.

What if the learning agent was to select actions non-deterministically? If the temperature

is set high[7], the agent is bound to eventually explore all the possible games that it can play given the rigid responses of the opponent. During this exploration, however, it will be losing a vast majority of games as the high temperature is forcing it to ignore the knowledge acquired in the process. Moreover, it will not become a better player as a result, because the nature of the binary goals does not recognize degrees of success. If we were to periodically drop the temperature to near-greedy levels for some intermediate test trials, we will witness an eventual convergence for such a subsequence of greedy trials. With the overall temperature of 1 and the greedy temperature of 0.1, such experimentation shows convergence of the naughts player after about 1800 games. What we have achieved, by raising the temperature, is a slightly more knowledgeable player, but not a more successful one in the sense of binary goals. Even with the high temperature, all the games still start with the cross placed in the center square as the crosses side always makes the first deterministically optimal move.

4.5.1.2 Playing against the Naughts

In this experiment the naughts player was emulated and the crosses player had to learn. The deterministic greedy approach again resulted in convergence to a single game, except now it took 20 trials, two of which were won by the naughts side and all the others were terminated because of illegal moves. Figure 4.13 shows the resulting tie game played as well as two of the trials leading to it. The increased number of trials are due to the fact that the crosses player has to learn to correctly place one piece more than the naughts player.

In these two experiments our DQ agents learned to play against deterministic opponents. In both cases the learned games were very rigid. One might wonder as to why we even need the 22k of memory for such a seemingly simple task. We have to remember, however, that the DQ-learning algorithm is still based on the Q-learning assumptions and expects

[7]A temperature of 1 ensures an almost random action selection for tic-tac-toe with the kind of levels of reinforcement used in our experiments.

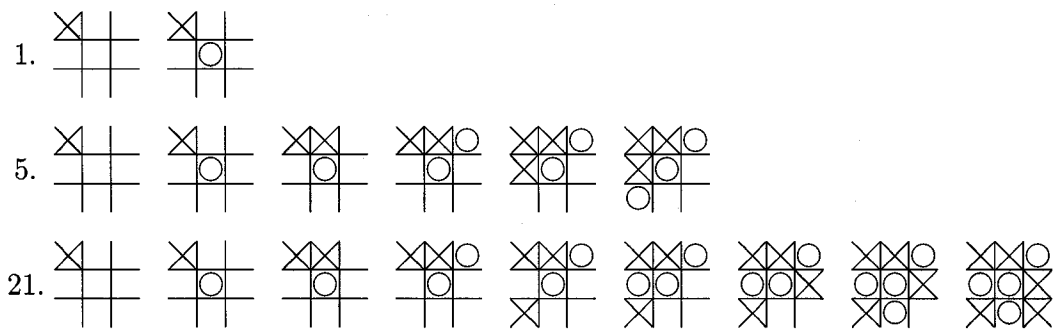

Figure 4.13: Trials 1, 5 and the first successful game 21 against the naughts.

a deterministic environment for convergence. Can we adapt the DQ-learning to the non-deterministic case in the same way that the Q-learning is adapted? The next section attempts to answer this question, and leads to unexpected results.

4.5.2 Non-deterministic DQ-learning

Let us recall how the Q-learning deals with non-determinism in the environment. When an old Q value, $\hat{Q}_{n-1}$, must be updated with a new estimate $\hat{Q}$, we don't simply replace the old estimate by the new. Instead, we move the old estimate towards the new in a convex manner. The amount of the adjustment is often described by a *learning rate* or *step size* parameter, usually denoted by α:

$$\hat{Q}_n \leftarrow (1-\alpha_n)\hat{Q}_{n-1} + \alpha_n \hat{Q}.$$

We now recall how the Q estimates were updated in the deterministic DQ-learning. There are only two possibilities: either the initial 0 is replaced by a -1, or -1 is replaced back by a 0! Under what circumstances would we ever need to switch from -1 back to 0? Such a switch would mean that a certain action, previously discovered as leading to failure, now appears to be, once again, acceptable. Specifically, the trajectory containing this action was previously discovered to be unacceptable, and then later deemed acceptable again. The implication

is that a Markov environment responds differently to the same action, and while one such response leads to a failure, another does not. If a given action results in an unreliable response from the environment, can such an action be considered acceptable? In other words, can an acceptable trajectory contain actions which sometimes lead to failure? We must answer in the negative. The definition of a binary goal does not allow for conditionally acceptable trajectories. The definition of the feedback signal, likewise, does not allow for probabilistic acceptability.

Our conclusion is that once a state-action pair is discovered to be unacceptable, it should remain so even if it sometimes leads to an acceptable trajectory due to the non-deterministic response of the environment. To successfully perform a *binary* task, the algorithm must find trajectories that are always acceptable. What if in a given environment there are no such trajectories, and if all that the agent can do is select the trajectory that is most likely (but not necessarily always) to be acceptable? We believe that such tasks are not binary and naturally bring us into the world of general n-ary goals, where *degrees of acceptability* are the only way to resolve this problem.

It should now be clear that our DQ-learning algorithm, when faced with a binary goal, does not require the use of the learning rate parameter and the corresponding gradual change in the estimate. In fact, we should now change our algorithm to prevent such "reversal" switches from -1 to 0. A ban on reversals not only ensures faster convergence, but guarantees it. Moreover, we can now attempt to learn binary tasks even for non-deterministic MDPs, and still represent the Q estimates using a single bit!

4.5.2.1 Learning to Play Tic-tac-toe against a Non-deterministic Opponent

Let us now confirm the convergence of the DQ-learning algorithm when learning to play tic-tac-toe against a non-deterministic opponent. In this experiment, we have set the temperature of the heuristic crosses player to be 1, in order to force it to play nearly randomly.

Most of the crosses plays will be weak, but because of sheer randomness it will also play a proportion of strong games. In contrast, we will set the temperature of the learning naughts player to 0, to make it always play to the best of its abilities. We can conclude that DQ-learning converges only when the opponent can no longer win any games.

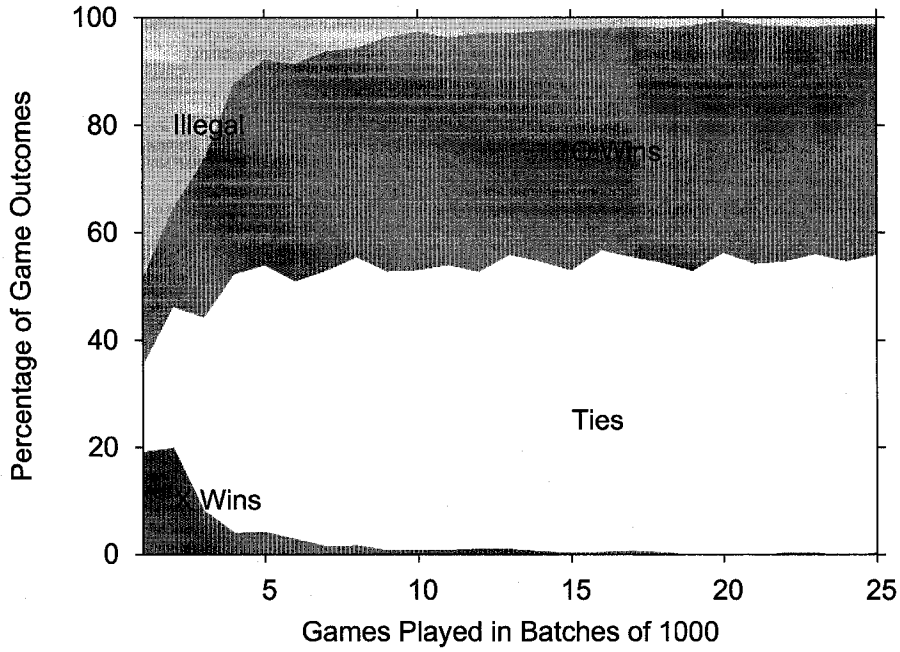

Figure 4.14: Convergence against a non-deterministic opponent with DQ-learning. Games shown in thousands.

Figure 4.14 shows the result of naughts playing 25,000 games against a nearly-random opponent. The horizontal axis shows the number of games played (in thousands) and the vertical axis shows the proportion of the four possible game outcomes: a win, a tie, a loss, or a premature game termination due to an illegal move. We can see that at the beginning, nearly half of the games end prematurely and the three other outcomes are equally likely. This is because both opponents initially play random games. The crosses side does so by design, whereas the naughts side has not learned anything yet. After playing about three

thousand games, however, we can see that the proportion of "undesirable" games (premature or losses) drops under 20% combined. After 10,000 games, the crosses manage to pull off, on average, only about 3 wins in a 1000. We must point out, however, that even after 50,000 games (not shown on the plot), the crosses still manage to win an odd game (albeit the chances of that are now 1 in 10,000). This simply means that there still exists an odd board position that the algorithm has not seen enough times to determine a correct move, but as more and more games are played such "gray" areas quickly disappear. After watching the learning agent play about 200,000 games and observing that it had not lost a single game in the past 100,000 of them, we can be sure, with a very high degree of certainty, that the algorithm has fully converged.

We have used a high temperature for the crosses side in this experiment because we wanted our learning agent to explore all kinds of board positions, and, thus, become proficient at playing not only tough opponents but also *any* opponent. After having witnessed the convergence described above, we can state with the same high degree of certainty that our DQ-learning agent learned *the game* of tic-tac-toe[8], and will not be beaten by any opponent! It has truly learned to perform the binary task of playing tic-tac-toe using a single bit to represent Q estimates and without knowing the rules of the game!!

4.5.2.2 Concluding Remarks

In this section we explored the potential of using the feedback signal — an alternative to reinforcements — as a method of supplying goal information to our learning agents. We have proposed DQ-learning as a discretized learning algorithm and shown it to be successful in using the feedback signal for task specification. Specifically, our DQ-learning was able to learn the binary tasks of playing tic-tac-toe against the heuristic-guided deterministic crosses and naughts opponents. Each table entry in the algorithm required a single bit of

[8]From the naughts point of view, of course.

storage, thus, experimentally confirming the conclusions of Section 3.4.3. Next, we introduced a source of non-determinism in the heuristic tic-tac-toe opponents, and showed how DQ-learning can be adapted to learn such non-deterministic binary tasks. As it turns out, even in the non-deterministic case, only a single bit per Q table entry is required to deal with binary tasks in Markov environments!

4.6 Solving the LightsOut Puzzle Using DQ-Learning

In the previous section, we demonstrated that significant memory savings can be realized if the goal arity is known. In this section, we show how the benefits of such savings can be attained in practice by applying DQ-learning to a series of puzzles collectively known under a brand name of "LightsOut".

While analytical methods of solving a variant of this puzzle are known, we proceed under a different premise of putting the learning agent in a much more difficult position of having virtually no initial information about the rules of the puzzle, and of not being able to perceive the symmetries inherent in the two-dimensional puzzle grid. In particular, without the described savings, a standard and already thrifty[9] Q-learning solution to a 5×5 variant of the puzzle would require at least $3\frac{1}{8}$ gigabytes of RAM with 4 bytes per table entry. Since our DQ-learning based solution required only 5 bits per table entry, this allowed us to solve the puzzle with a 1 gigabyte machine available to us at the time of this writing.

Finally, we also consider a variant of this puzzle for which no analytical or algorithmic solutions are known. Our DQ-learning algorithm is equally applicable to the new puzzle and can find solutions to such a puzzle with exactly the same amount of effort, while still not knowing the rules of this puzzle at the outset.

[9]Single rather than double precision floating point Q values.

4.6.1 History of the LightsOut Puzzle

LightsOut is an electronic puzzle game originally manufactured by Tiger Electronics[10]. The puzzle has been favourably received by puzzle lovers throughout the world with several web sites dedicated to it[11]. In its original form, the puzzle consists of a 5×5 grid of lights, which also act as buttons. Each light can be either "on" or "off". Pressing on a light will change its state along with each of the four immediate neighbour lights. Figure 4.15 [12] shows the three types of light neighbourhoods resulting in pressing a light in the middle of the board (left), in the corner (center), and on the edge (right). The larger cross identifies the light being pressed, while the smaller crosses identify the other affected lights.

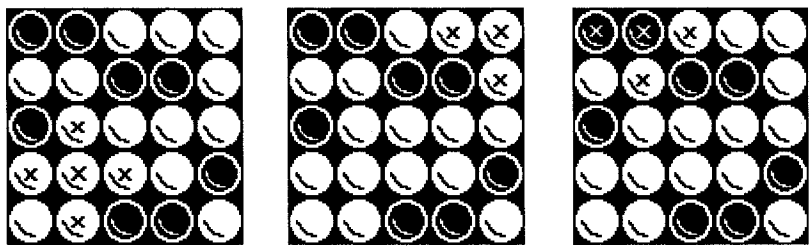

Figure 4.15: LightsOut puzzle with classic neighbourhoods. The larger crosses identify the pressed lights and the smaller crosses identify the other affected lights.

The lights on the edge of the grid, and in the corners, clearly have fewer affected neighbours. There exist, however, a variation of the puzzle where the opposing edges of the grid are connected, making the grid into a veritable torus. This modification is also used in a

[10]Tiger Electronics was acquired by Hasbro in 1998. Some years earlier, back in 1983, another company by the name of Vulcan Electronics also manufactured a puzzle called XL-25 which can be considered as a precursor to LightsOut.

[11]The reader is invited to visit two comprehensive sites currently residing at `www.haar.clara.co.uk/Lights/index.html` and `www.geocities.com/jaapsch/puzzles/lights.htm`

[12]Image taken from a Palm OS adaptation of the puzzle by Andy Scheffler.

4×4 key chain version of the puzzle sold by Tiger. Figure 4.16 shows the "torus" effect: the lights on the edges and in the corners also have 4 neighbours some of which appear on the opposite edge due to the wrapping effect. It can be shown [8] that the "torus" modification only makes the puzzle somewhat more challenging.

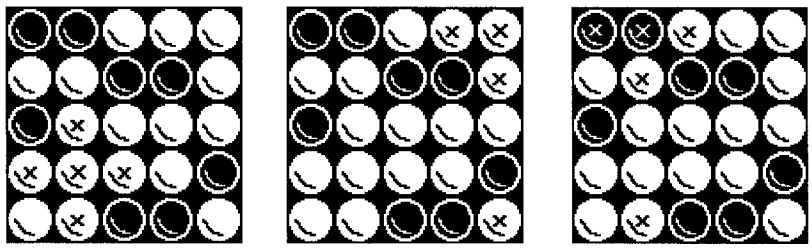

Figure 4.16: LightsOut puzzle with "torus" neighbourhoods. The center and right boards show additional neighbours.

The objective of the game is to turn off all the lights (hence the name of the puzzle) in the shortest number of moves, starting from one of many possible initial light patterns.

There are many other known variations of the puzzle. For example, it is clear that the grid dimensions can be easily changed. In this chapter, we consider puzzles of increasing grid sizes. Other possible variations include changing the configuration of the neighbourhood of affected lights, changing the number of states of a light (implemented in the LightsOut 2000 version, where each light can be either red, green, or off) or bringing the puzzle into the third dimension as in the LightsOut Cube.

4.6.2 Analyzing LightsOut

The LightsOut puzzle has several appealing mathematical properties responsible for attracting the attention of mathematicians in the recent years. It serves as an exciting demonstration of linear algebra methods that are useful in solving a puzzle with a sizeable international fan following. First of all, it turns out that if a solution to the puzzle exists (given some

initial light pattern), the *order* in which the lights are pressed in such a solution *does not matter*. Furthermore, since pressing the same light twice in a row leaves all the lights in their original states, it is clear that any solution to the puzzle will either require a certain light to be pressed once, or not at all. This, in turn, means that every solution to the puzzle is a pattern of lights to be pressed on the same grid, and that we only need to consider such patterns as potential solutions. Some variants of the puzzle, such as the 4×4 key chain version mentioned above, can be solved from any initial pattern of lights with an appropriate pattern of light presses, i.e. the puzzle state space is fully connected. This is not the case, however, for other variants such as the 5×5 version.

How can we then find the solution pattern if it exists? The detailed account of the solution methodology can be found in [8], [77], and [44]. For a 5×5 variant, we can consider the grid as a state vector of 25 binary values, where 0 means "off" and 1 means "on". An action of pressing on a single light (or several lights, for that matter) can similarly be represented by a vector of 25 bits, where 1's indicate the lights which will change their state. The result of pressing on a light is obtained as the sum of the two vectors modulo 2. Alternatively, this can be viewed as an XOR operation on 25 bit words.

Since such a summation is commutative, the order in which we add the "action" vectors does not matter, thus, confirming the earlier stated property that the order in which the lights are pressed is also irrelevant. In matrix form[13], the solution vector $\vec{x}$ is, thus, related to the initial pattern $\vec{p}$ as in:

$$\vec{0} = \vec{p} + \vec{x}A,$$

where $\vec{0}$ is a vector of all 0s (the desired state with all lights off), and A is a matrix of individual "action" vectors, where a_{ij} is 1 if light i is changed by pressing on light j, and 0 otherwise. Solving for $\vec{x}$ we get:

$$\vec{x} = -\vec{p}A^{-1},$$

[13]All vectors are row vectors.

where A^{-1} is the inverse of matrix A modulo 2. Note that in modulo 2 arithmetic, a negation of a vector is exactly the same as the original vector, and so $\vec{p} = -\vec{p}$. We can thus interpret A^{-1} as the matrix where the i^{th} row represents a pattern of buttons to be pressed in order to switch the state of the i^{th} light alone, leaving all the other lights unaffected. If the matrix A can be inverted, it would mean that each light is independent of the others in the sense that it can be toggled on or off individually, with the right combination of buttons.

Let's consider an example of the 4×4 torus variant. The corresponding matrix $A_{4\times4}$ is shown in Figure 4.17. Curiously, its inverse is exactly the same as $A_{4\times4}$ itself, i.e. $A^2_{4\times4} = I$, where I is the identity matrix. This is not the case in general.

The existence of the inverse guarantees the existence of a solution to any initial light pattern. For example, to solve the puzzle for the pattern shown in Figure 4.18, we need to press on the lights that are part of the solution vector

$$\begin{aligned}&[1\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 1\ 0\ 0\ 1] \times A_{4\times4} \\ &\quad = [0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 1]\,.\end{aligned}$$

Unfortunately, for many of the puzzle variants, the matrix happens to be singular, i.e. no inverse exists. This would mean that not every light pattern is solvable. All is not lost, however, as in such cases we can determine the so-called pseudo-inverse, by using the Gaussian elimination algorithm, for example. By multiplying a matrix by its pseudo-inverse, we would not get an identity, but a matrix which contains an identity sub-matrix. For instance, the 3×4 variant has a pseudo-inverse, and the resulting "near-identity" matrix is shown in Figure 4.19. Note that four rows of this matrix contain all zeros. This tells us that four rows of the original matrix $A_{3\times4}$ are linearly-dependant on the other eight rows. Hence the rank of A is also 8. Knowledge of the rank gives us important information with regards to the solvability of the puzzle variant. Specifically, for an $m \times n$ puzzle, the fraction of solvable initial light patterns can be expressed as:

$$\frac{1}{2^{mn-r}},$$

$$\begin{pmatrix}
1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1
\end{pmatrix}$$

Figure 4.17: Matrix A for the 4×4 torus LightsOut puzzle. Note that in this case $A^{-1}_{4\times4} = A_{4\times4}$.

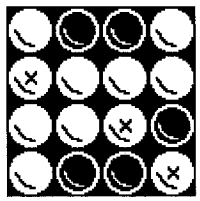

Figure 4.18: Solution to a 4×4 variant. Buttons to be pressed are marked with crosses.

where r is the rank of the corresponding puzzle matrix. We can, therefore, predict that only $\frac{1}{2^{12-8}} = \frac{1}{16}$, or about 6% of the total $2^{12} = 4096$ light patterns are solvable for the 3×4 puzzle.

The row vectors in the pseudo-inverse that correspond to the zero rows in the "near-identity" matrix are called *quiet* button patterns, in that they do not affect any light pattern to which they might be applied. The pseudo-inverse of the 3×4 puzzle, for example, gives us the four quiet patterns shown in Figure 4.20. These are called the *generators* of the quiet patterns, because when combined together, they produce other quiet patterns, all of which collectively form the so-called *null* space of the puzzle. In fact, the four original quiet patterns form the basis of the null space and their number establishes the dimension of the null space, which, in this case, is equal to 4. If we find an arbitrary solution to some light pattern, we can combine it with any quiet pattern to produce another solution, consisting of potentially fewer steps. By examining all such combinations, we can choose the shortest solution.

While analytical methods of solving LightsOut are known in some settings, we proceed under a different premise of putting the learning agent in a much more difficult position of having virtually no initial information about the rules of the puzzle, and of not being able to perceive the symmetries inherent in the two-dimensional puzzle grid. These are the same extreme conditions of no prior knowledge that were in effect in [13]. Our approach is essentially that of a modified reinforcement learning scheme. While a straight-forward

$$\begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}$$

Figure 4.19: Near-identity matrix for a 3×4 variant of LightsOut. It is the result of $A_{3\times4} \times \hat{A}^{-1}_{3\times4}$, where $\hat{A}^{-1}$ denotes the pseudo-inverse.

Q-learning implementation would require 4 bytes per Q table entry on 32 bit processors, our DQ-learning modification better utilizes the available memory, using only 5 bits per table entry. We describe our approach in the following section.

4.6.3 Solving LightsOut Using DQ-learning

The DQ-learning algorithm was already described in Chapters 3 and Section 4.5 as an adaptation of classical Q-learning to the feedback signal. To apply it to a new domain of LightsOut puzzles, we must specify the parameters of the learning task at hand and, specifically, determine the goal arity. The following two sections consider somewhat different definitions of what constitutes a successful solution of the puzzle. For each grid size, we report on experiments conducted with both the classic and "torus" variants of the puzzle.

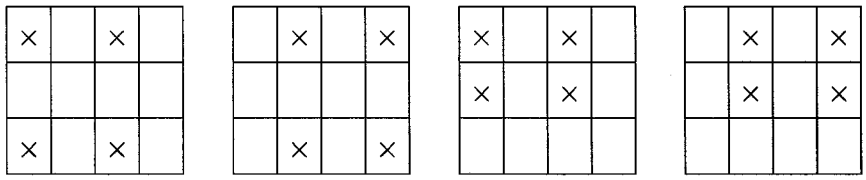

Figure 4.20: Quiet patterns for the 3×4 variant extracted directly from the pseudo-inverse. The buttons to be pressed are denoted with crosses.

4.6.3.1 The LightsOut Puzzle as a Binary Goal Problem

The LightsOut family of puzzles requires reaching a certain goal state (all lights are off) from some initial light pattern. Any sequence of actions that leads to the goal state can be considered as one of several possible solutions to the puzzle. Based on the earlier analysis, it is clear that if a solution exists, there must be infinitely many such solutions. So we should not be satisfied with finding just any solution, especially in a 4×4 "torus" variant of the puzzle (or any other variants with similar properties) where every single pattern of lights is solvable. It is natural to prefer, in general, shorter solutions to longer ones, with the shortest possible solution being the most attractive. When some acceptable solutions are preferable to others, and especially when we are looking for the shortest path to goal, we enter the domain of n-ary goals.

Can the LightsOut puzzle be specified as a binary goal — the goal of simplest arity? Since we don't know, in general, what is the length of the shortest solution, we cannot designate the latter as the only acceptable solution. The elegant mathematical properties of the puzzle, however, allow us to put a simple upper bound on the shortest solution regardless of the starting light pattern. Since successive presses on the same light cancel each other's effects out, the shortest solution must either contain a single press of any given light, or no press at all. This means that on an m by n grid, the shortest solutions to any initial pattern is at most mn. For instance, on a 5×5 grid the upper bound is $5 \times 5 = 25$ light presses.

Given this simple analysis, we can now formulate a binary goal for the $m \times n$ LightsOut

puzzle. A trajectory is considered acceptable if it starts with the initial light pattern and ends in the goal state in at most nm light presses, and unacceptable otherwise. It is important to underscore that for such a binary goal, we are satisfied with any solution of length $\leq mn$, and not just the shortest solution.

Since DQ-learning is essentially a discretized version of the ordinary Q-learning algorithm, it has similar convergence properties. In particular, it is only proven to converge if the environment dynamics and the reinforcement signal have the Markov property. In the case of the binary goal described above, the Markov property no longer holds for the reinforcement signal as the same light pattern and press combination may be penalized differently, depending on how many actions were already taken in the past. To ensure that the algorithm can converge, we must introduce an additional artificial state variable, i.e. the count of actions taken so far, to restore the Markovian nature of the reinforcement signal. As it will be evident in Section 4.6.4, this, curiously, makes the learning task more difficult for our DQ-learning algorithm as compared to the n-ary goals that we will consider in that section.

4.6.3.2 Æip Setting

The overall Æip setting for our experiments is straightforward. The LightsOut world has $mn + 3$ outports, all of which save one (End Of Episode signal, or EOE) are fed into the corresponding inports of the DQ-learning entity. Each light of the puzzle world has a corresponding outport with possible values of 0 (light is off) and 1 (light is on) making up the first mn outports. In addition to these, we have the reinforcement outport corresponding to the original feedback signal, the action count outport required for convergence, and the EOE outport needed by the interaction manager component to determine when the current learning trial ends.

For binary goals, the reinforcement will be 0 until a failure is discovered whereupon a

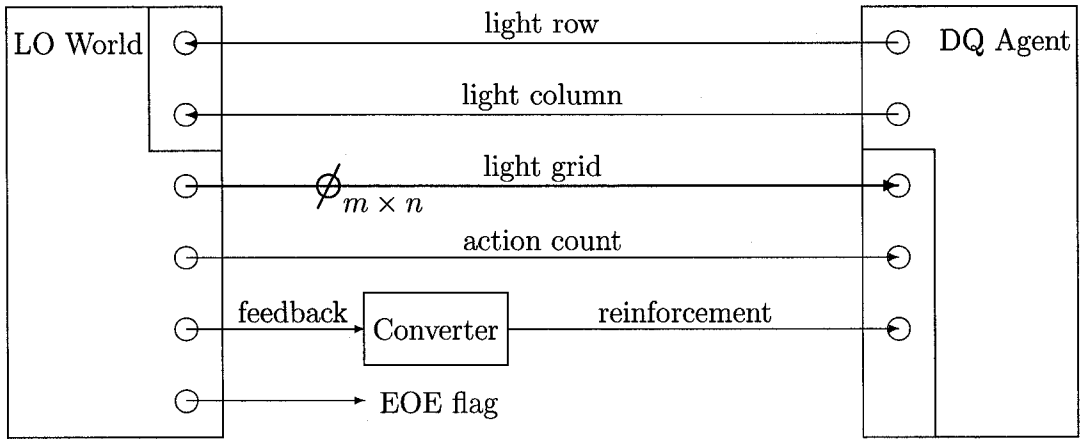

Figure 4.21: Æip configuration for LightsOut.

penalty of -1 will be signaled. At this point the EOE boolean outport is set to 'true' to signal the end of the learning trial. On the other hand, the failure is discovered when the action count reaches the upper limit equal to mn for an $m \times n$ grid of lights. If the goal pattern (all lights are off) is reached before this limit, the EOE is also set to 'true' to signal a successful termination of the learning trial.

The DQ-learning entity will have $nm + 2$ inports as discussed above, and two outports corresponding to the row and the column index of the light to be pressed next. The resulting configuration is shown in Figure 4.21. In case of the n-ary goals discussed later, the sole difference in the diagram will be the lack of the action count inport in the DQ-learning agent. The world entity does not have to be configured differently — its action count outport will simply remain unconnected, and thus unused. *This also highlights the feature of Æip of not requiring all outports to be used.*

4.6.3.3 Taking Advantage of Symmetries

It is not difficult to see that every light pattern in LightsOut has several equivalent patterns induced by the symmetries inherent in the two-dimensional nature of the puzzle. In

the straightforward configuration shown above, however, the agent treats all such equivalent patterns as different cases and must learn their solvability independently. This is not an oversight on our part as we intentionally want to provide the agent with as little information as possible and let it, on its own, discover and exploit any useful relationships (e.g. symmetries) that may exist between different environment states. Unfortunately, the Q-learning (and thus DQ-learning) is known to be a so-called model-free (and, consequently, generalization-free) algorithm, simply incapable of discovering such relationships. In the light of this property of DQ-learning, it is interesting to see how we can augment the Æip configuration so that the information about symmetries is implicitly supplied.

Figure 4.22 shows one way in which this can be accomplished. The particular attractiveness of this solution stems from the fact that we do not need to modify either the LightsOut World entity, or the DQ-learning Agent entity. All we have done is inserted a middleware entity, called the Symmetry Map in the diagram. The purpose of this entity is to consistently map all the equivalent symmetrical light patterns into always one and the same of these patterns. As a result, the agent is confronted with much fewer distinct world states and, therefore, can converge to a solution much faster. This approach also showcases the flexibility of Æip design, where standard entities can be reconnected and assembled into new experiments. We also wish to point out that this approach modifies the original puzzle dynamics apparent to the agent. We must, hence, take this into consideration when using algorithms which construct a model of the environment.

The Symmetry Map entity was implemented to encode light patterns as bit strings and map all equivalent patterns onto the one with the smallest binary code. The row and column of the action selected by the agent must also be mapped according to the same transformation, but in the reverse direction. Let us now consider how many equivalent patterns exist due to symmetry. If we are dealing with an $m \times n$ grid, where $n \neq m$, i.e. a non-square grid, each pattern has three symmetrical patterns that can be obtained by either rotating the grid 180 degrees, or reflecting about a vertical axis (mirror image), or combining

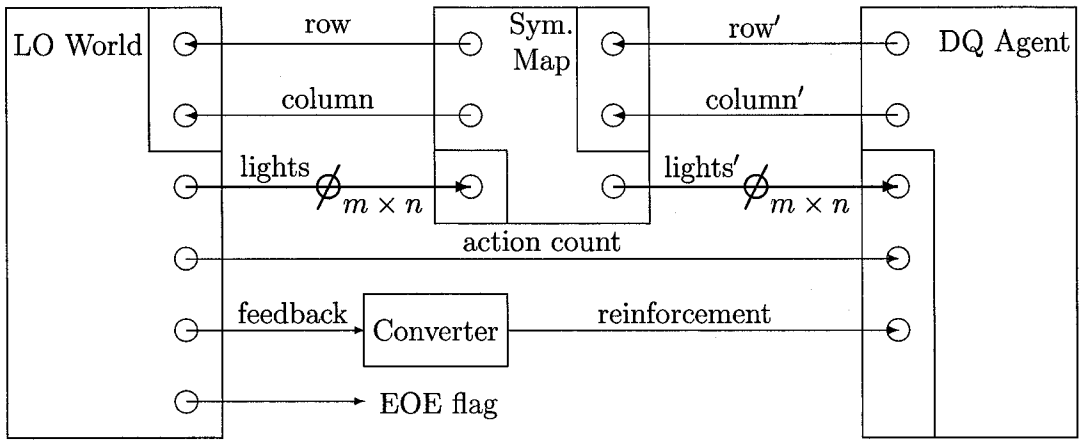

Figure 4.22: Æip configuration for LightsOut with a Symmetry Map entity.

the rotation and reflection. Given a square grid, we can also rotate the pattern by 90 and 270 degrees and apply reflection to both. This gives us a total of 4 patterns per equivalency class for a non-square grid and 8 patterns for a square grid.

Curiously, we can do even better in the case of the "torus" variant of the puzzle. Since the edges of the grid are connected, we can safely add another transformation, namely, a shift along either of the two dimensions. Moreover, we can shift the pattern by either one, two or any number up to the size of the grid along either of the two dimensions. Such shifting alone gives us $m \times n$ equivalent patterns. If combined with reflection and rotations we can produce equivalence classes of considerable size (e.g. $5 \times 5 \times 8 = 200$ patterns for a 5×5 puzzle) and realize dramatic reductions in convergence time.

We must remember that some patterns would, of course, have the property that some or all of their symmetrical counterparts might turn out to be the same. An extreme example would, naturally, be the pattern where the lights are either all on or all off. It is not difficult to see, nevertheless, that we should be able converge much faster taking into the account the symmetries. Where applicable, we will contrast this dramatic difference in convergence time in the reported experiments.

We have conducted a number of experiments with our definition of the binary goal for several puzzles of increasing grid size. We decided to report the results for 4×4 (16 lights) and 3×7 (21 lights). Due to enormous memory requirements[14] for the 5×5 puzzle as a binary goal, we were only able to solve it using n-ary goals as discussed in Section 4.6.4.

4.6.3.4 Solving 4×4 LightsOut using Binary Goal

For a 4×4 grid, we have 16 lights and thus 16 different actions that the DQ learner can select from at each step. Interestingly, the "torus" variant of this puzzle is known to be fully connected, meaning that we can find a solution of length at most 16 for any initial light pattern. This is not the case for the classic variant. The puzzle has 2^{16} different light patterns, each of which can occur for different values of action count which itself ranges from 0 to 16 (17 different values, if we include 0). Since the number of possible actions is also 16, the Q table for this learning task has $2^{16} \times 17 \times 16 = 17,825,792$ entries in total. According to the DQ-learning algorithm, each of these entries requires a single bit of storage for a binary goal. Thus, we have a combined memory requirement for our DQ learner of $17,825,792 \div 8 = 2,228,224$ bytes or $2\frac{1}{8}$ megabytes. An ordinary non-discretized Q-learning algorithm would have required at least 32 times more memory, or more than 68 megabytes!

Figure 4.23 shows the result of a typical experiment where after about 8 million learning trials (or attempts), a solvability plateau of just over 6% is reached, meaning that an acceptable solution is found for only a small portion of the $2^{16} = 65536$ different starting light patterns. This number also implies that an average of approximately 125 attempts was spent on each initial light pattern before the pattern was fully solved, or determined to be unsolvable.

Since no solution was found at depth 16 for the apparently unsolvable patterns, it means

[14]Over $2\frac{1}{2}$ gigabytes already taking advantage of the memory savings offered by DQ-learning, i.e. a single bit per table entry!

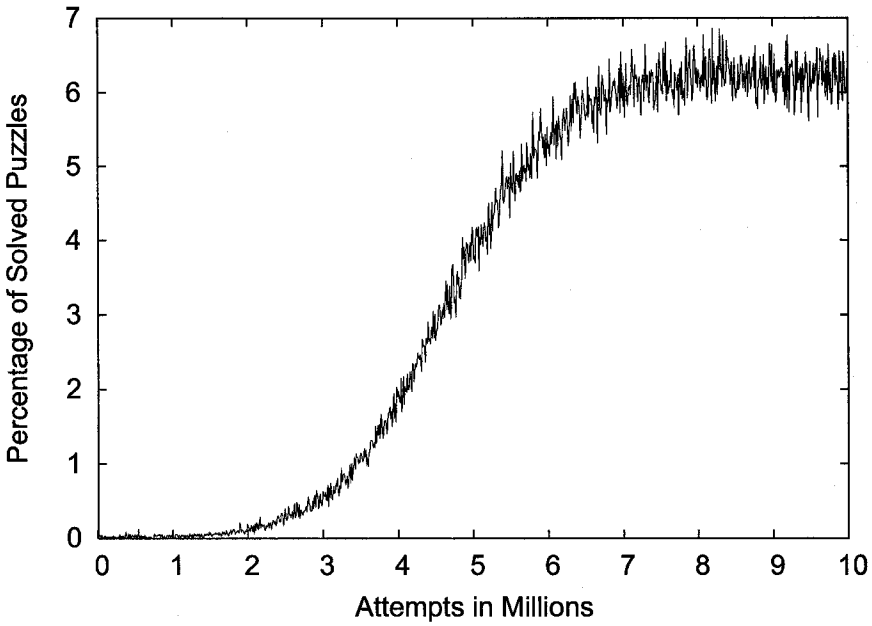

Figure 4.23: Solving classic 4×4 LightsOut as a binary goal using DQ-learning.

that no solution exists. Since the original Q-learning algorithm is proven to converge, its discretized version in the form of DQ-learning will too, as it does not affect the reasoning of the proof. The proof works under the assumption that each of the state-action pairs are visited a sufficient number of times. Since the initial light patterns are selected at random with a uniform distribution, we can expect each state-action pair to be visited sufficient number of times. We, thus, have no reason to doubt the empirical result that only about 6% of patterns are solvable.

Given the analytical apparatus introduced earlier, we can calculate the fraction of solvable patterns. The pseudo-inverse of the puzzle matrix for the 4×4 classic variant has a rank of 12, which means that the dimension of the null space works out to be $4 \times 4 - 12 = 4$. Therefore, the fraction of solvable patterns works out to be $\frac{1}{2^4} = \frac{1}{16} = 6.25\%$, confirming the above empirical result.

Figure 4.24 on the other hand, shows the result of a typical experiment for a "torus" variant where after about 15 million learning trials, the puzzle was fully solved, reaching a 100% solvability. This means that an acceptable solution is determined for every one of the $2^{16} = 65536$ different starting light patterns. This number also implies that an average of approximately 250 attempts was spent on each initial light pattern before the pattern was fully solved.

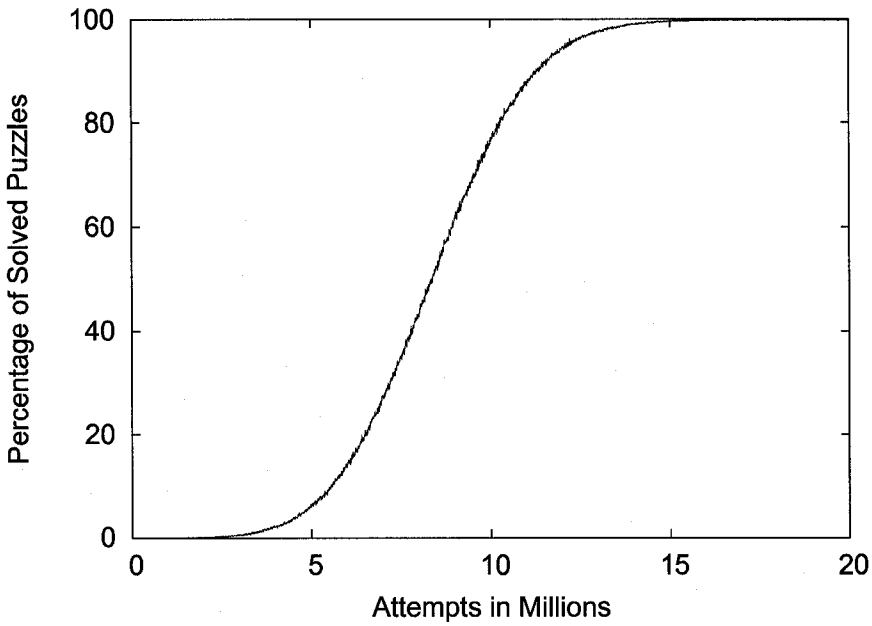

Figure 4.24: Solving torus 4×4 LightsOut as a binary goal using DQ-learning.

4.6.3.5 Solving 3×7 LightsOut using Binary Goal

For a 3×7 grid we have 21 lights, and thus 21 different actions that the DQ learner can select from at each step. Unlike the 4×4 case, the "torus" variant of this puzzle is not fully connected, meaning that only a fraction of initial light patterns have solutions. In contrast, the classic variant for this grid size has a solution for every initial light pattern.

The puzzle has 2^{21} different light patterns, each of which can occur for different values of action count, which itself ranges from 0 to 21 (22 different values, if we include 0). Since the number of possible actions is also 21, the Q table for this learning task has $2^{21} \times 22 \times 21 = 968,884,224$ entries in total. As before, each of these entries requires a single bit of storage for a binary goal. Thus, we have a combined memory requirement for our DQ learner of $968,884,224 \div 8 = 121,110,528$ bytes or $115\frac{1}{2}$ megabytes. An ordinary non-discretized Q-learning algorithm would have required at least 32 times more memory, or more than 3.6 gigabytes of RAM! This is already outside of the reach of the computers that were available to us at the university's graduate lab at the time of this writing, proving that the benefits of DQ-learning are quite tangible in making a difference between being and not being able to tackle a certain learning task.

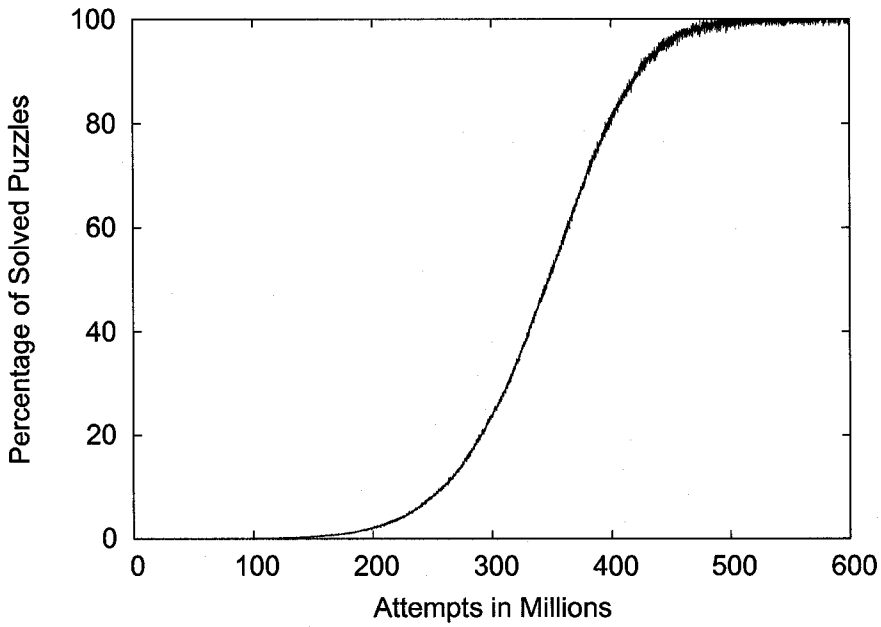

Figure 4.25: Solving classic 3×7 LightsOut as a binary goal using DQ-learning.

Figure 4.25 shows the result of a typical experiment where after about 600 million (loga-

rithmic scale, again) attempts, the puzzle was fully solved, reaching a 100% solvability. We can see that, on average, about 286 attempts per each different light pattern were required to fully solve the puzzle.

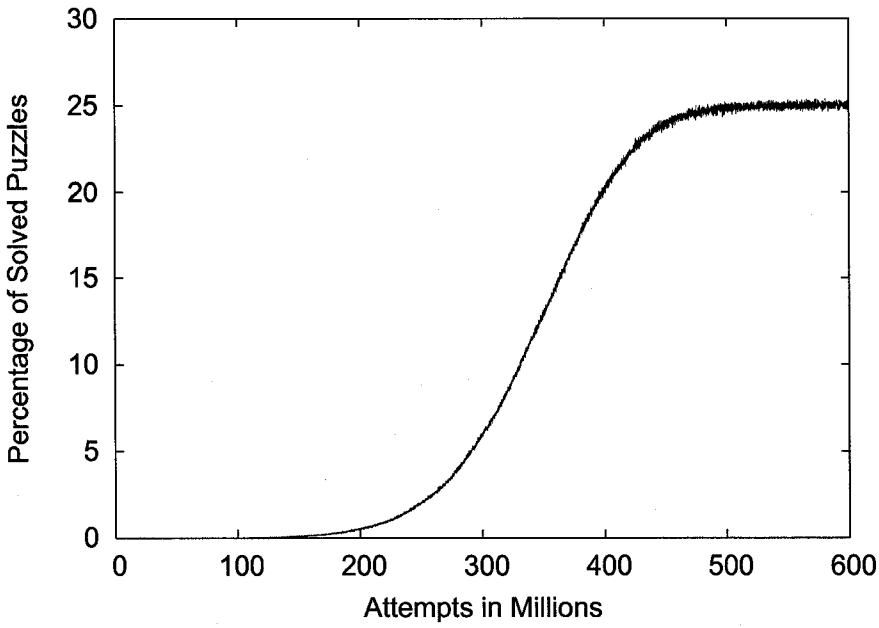

Figure 4.26: Solving torus 3×7 LightsOut as a binary goal using DQ-learning.

Figure 4.26 on the other hand, shows the result of a typical experiment with the torus variant for the same grid size, where, similarly, after about 600 million attempts, the puzzle was fully solved. Note that only a quarter of all initial patterns appear to be solvable. Since no solution was found at depth 21 for these, it means that no solution exists. The pseudo-inverse of the puzzle matrix for the 3×7 torus variant has a rank of 19, which means that the dimension of the null space works out to be $3 \times 7 - 19 = 2$. Therefore, the fraction of solvable patterns works out to be $\frac{1}{2^2} = 25\%$, confirming the above empirical result.

4.6.4 LightsOut Puzzle as an N-ary Goal

In the case of binary goals, we only imposed an upper bound on the length of acceptable solutions, but did not require them to be the shortest. What if we were to accept only the shortest solutions and reject all the others? Theoretically, this is still a binary goal. We would need to know by examining the current solution alone, whether it is the shortest or not. As was discussed in Section 3.4.6, we would not be able to make such a determination, in general, and such goals are termed *exploratory.* We can, however, achieve a compromise, whereby we specify an n-ary[15] goal instead, preferring shorter solutions to longer ones. Since every solvable initial light pattern will have infinitely many different solutions, with potentially very long trajectories, we still need to impose an upper bound on acceptable solutions. The lower the limit, the fewer bits we would need to allocate per Q table entry.

Earlier, we established that for the $m \times n$ puzzle, the shortest solution cannot be longer then mn. This is precisely the acceptability threshold we are going to choose here too. For example, for a 4×4 puzzle we are going to prefer solutions of length 0 (when the initial pattern has all the lights off), then solutions of length 1, then — of length 2, and so on, until we reach solutions of length 16, which is going to be the longest acceptable length — all longer solutions will be unacceptable and constitute a failure. We, thus, categorize all trajectories into 18 classes, 17 of which are the acceptable solutions of lengths 0 to 16, and the one class which contains all the unacceptable solutions. The arity of such a goal, therefore, is 18.

Given such a definition of the goal, the dynamics of the corresponding feedback signal would have the Markov property, because the best course of action (i.e. towards the shortest solution) from the given light pattern does not depend on how we have arrived at this pattern, and, in particular, in how many steps. This eliminates the need for the action count as the

[15]Here n is simply part of the historical name of the goals of arity greater than 2, and is not related to the column dimension of the light grid used elsewhere.

extra state variable we used for binary goals.

The Æip setting for our experiments is almost exactly the same as with binary goals, except that the action count outport is no longer needed; the LightsOut world still has such an outport, but the corresponding signal is not being fed into the DQ learner. Similarly, the algorithm must be supplied with the appropriate goal arity, in order to be able to set aside enough bits per Q table entries.

4.6.4.1 Solving 4×4 LightsOut using an 18-ary Goal

As discussed above, we can find the shortest solutions in a 4×4 puzzle with a goal of arity 18. Since this puzzle is fully connected, we will find the shortest solution for every initial light pattern. Since the action count state variable is no longer taken into consideration, the number of state-action pairs that make up the Q table can be calculated to be just $2^{16} \times 16 = 1,048,576$. Each of these entries must be able to store 18 different returns, requiring 5 bits of storage. The total memory requirement for the Q table is, thus, $1,048,576 \times 5 \div 8 = 655,360$ or only 640 kilobytes! In fact, with a crafty encoding[16] we can reduce this down even further to approximately 534 kilobytes, observing that we under-utilize the informational content of five bits. The reader should contrast this with the 4 megabytes that would have been required with the standard Q-learning.

Figure 4.27 shows the result of a typical experiment where after about $\frac{1}{2}$ million attempts the puzzle was fully solved, reaching the earlier confirmed 6.25% solvability. We can clearly see that it takes much longer to solve the binary task (at least 15 times longer). This should not be surprising, considering the fact that a binary task requires a much larger Q table, thus taking more time to converge to a correct value for each and every table entry. What is surprising, however, is that while the n-ary task of solving LightsOut seems more complicated than the corresponding binary task, in practice, the former requires less memory

[16]This encoding was not actually implemented for the sake of program simplicity.

and, as a result, offers faster convergence. We would have expected a more complex task to be solved with more effort. We surmise, that this effect is due to the model-free nature of the algorithm used. If DQ-learning were to be replaced with a model-based discretized algorithm capable of tackling non-Markovian environments, we would be likely to realize a benefit in choosing a simpler binary task over the more complex n-ary counterpart. We are of the strong opinion that, all other conditions being equal, algorithms designed to take advantage of the simplicity of the binary task in non-Markovian environments will have an edge over general algorithms designed to solve tasks of arbitrary arity. This is still a question to be investigated.

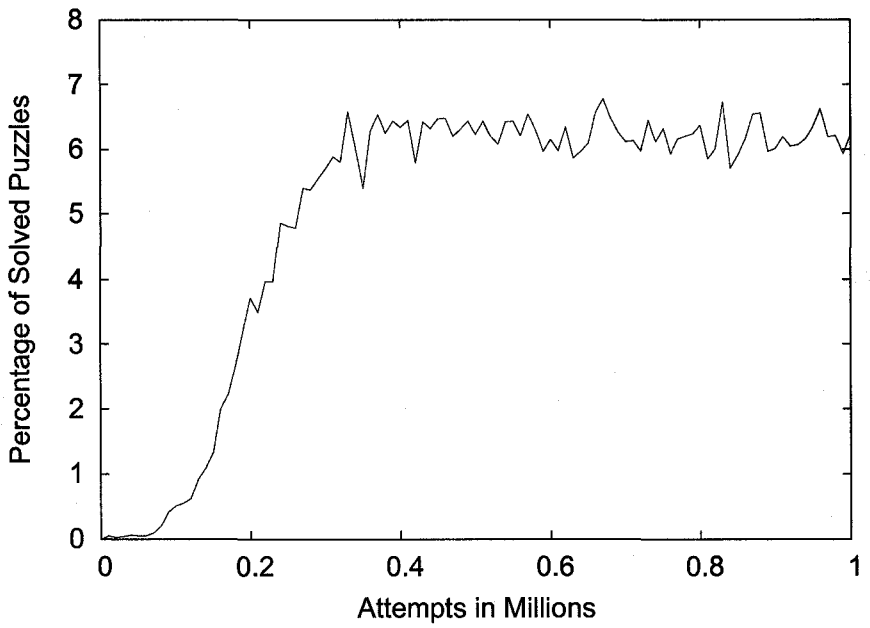

Figure 4.27: Solving classic 4×4 LightsOut as an n-ary goal using DQ-learning.

Figure 4.28 shows the result of a typical experiment with the "torus" variant, where after about 1.5 million attempts, the puzzle was fully solved, reaching 100% solvability. We can observe again that as a result of a smaller Q table, it takes much fewer attempts (about 10

times fewer) to solve the same puzzle as an n-ary task.

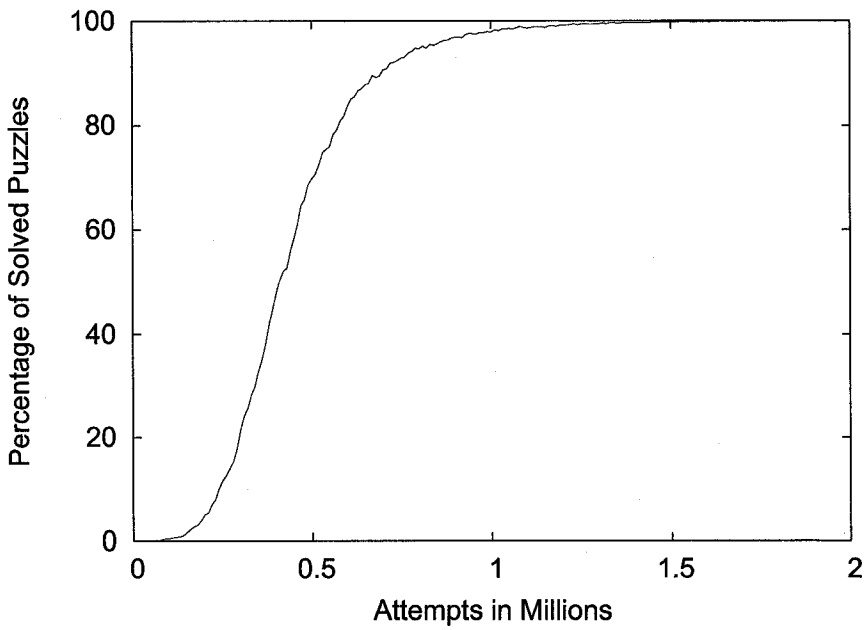

Figure 4.28: Solving torus 4×4 LightsOut as an n-ary goal using DQ-learning.

We have also tried the Æip configuration with the Symmetry Map for this "torus" variant, and Figure 4.29 shows two other curves resulting from learning experiments where we take advantage of the symmetry of the puzzle. The curve labeled "sym" shows the convergence of the learner when we are only using the reflection and rotation symmetries, while the curve labeled "shift sym" takes advantage of the "torus" shift symmetries as well. It is clear that the convergence speed is dramatically improved, namely, by *about 30 times*! Although not empirically verified, we believe that similar improvements with regard to convergence speed can be expected in the case of the classic variant of the 4×4 puzzle.

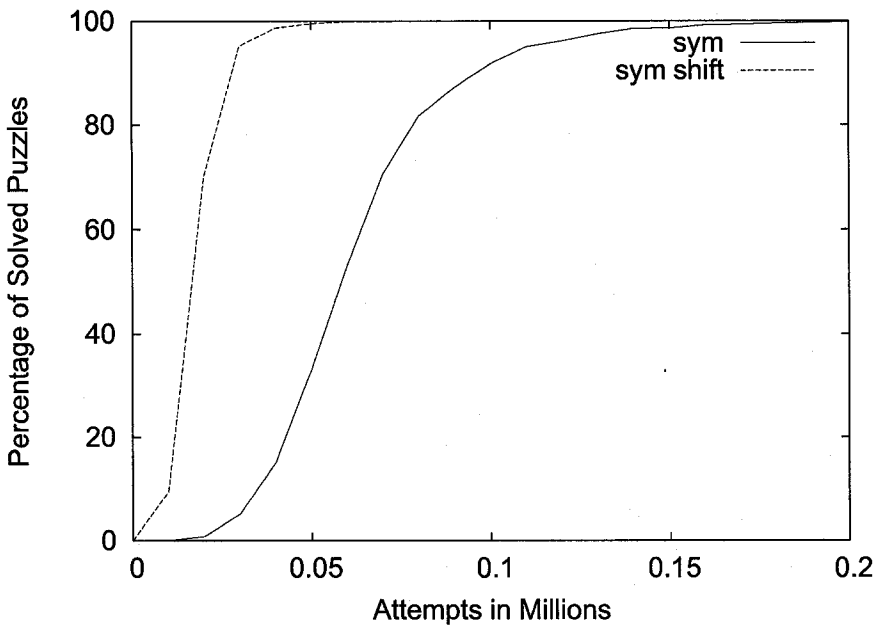

Figure 4.29: Solving torus 4×4 LightsOut as an n-ary goal using DQ-learning and taking advantage of reflection, rotation, and shift symmetries.

4.6.4.2 Solving 3×7 LightsOut using a 23-ary Goal

In general, finding shortest solutions for an $m \times n$ puzzle entails a goal of $mn+2$ arity. Thus, we need a goal of arity 23 for a 3×7 puzzle. We have already seen that the classic variant of this puzzle is fully connected, while only a quarter of patterns the "torus" variant can be solved. We expect to confirm these facts by solving the puzzle as a 23-ary goal problem. As before, we can calculate the size of the Q table as having $2^{21} \times 21 = 44,040,192$ entries in total. Each of these entries must be able to store 23 different returns, still requiring the same 5 bits of storage. The total memory required to store the Q table is, thus, equal to $44,040,192 \times 5 \div 8 = 27,525,120$ bytes or $26\frac{1}{4}$ megabytes. Again, the reader should contrast this with at least 168 megabytes needed by the Q table of an ordinary non-discretized Q-

learning algorithm.

Figure 4.30 shows the result of a typical experiment where after about 60 million attempts the puzzle was fully solved. This is yet another empirical confirmation of the 100% solvability of this variant.

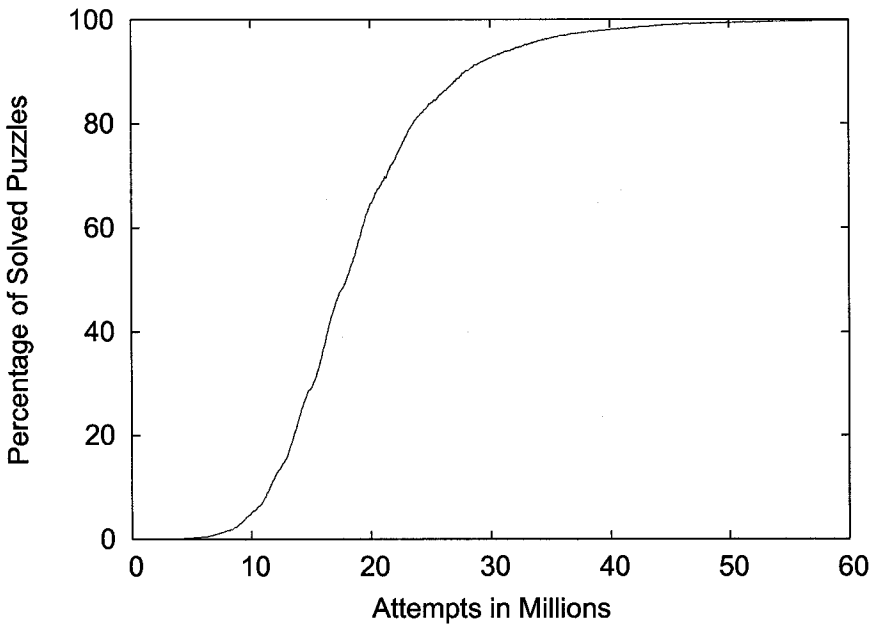

Figure 4.30: Solving classic 3×7 LightsOut as an n-ary goal using DQ-learning.

Figure 4.31 shows the result of a typical experiment with the "torus" variant, where after about 30 million attempts the puzzle was fully solved. The same plateau at the same level of 25% solvability was reached as in the case of the binary goal. We thus have another empirical confirmation that this variant of the puzzle is not fully connected, with only a quarter of light patterns being solvable.

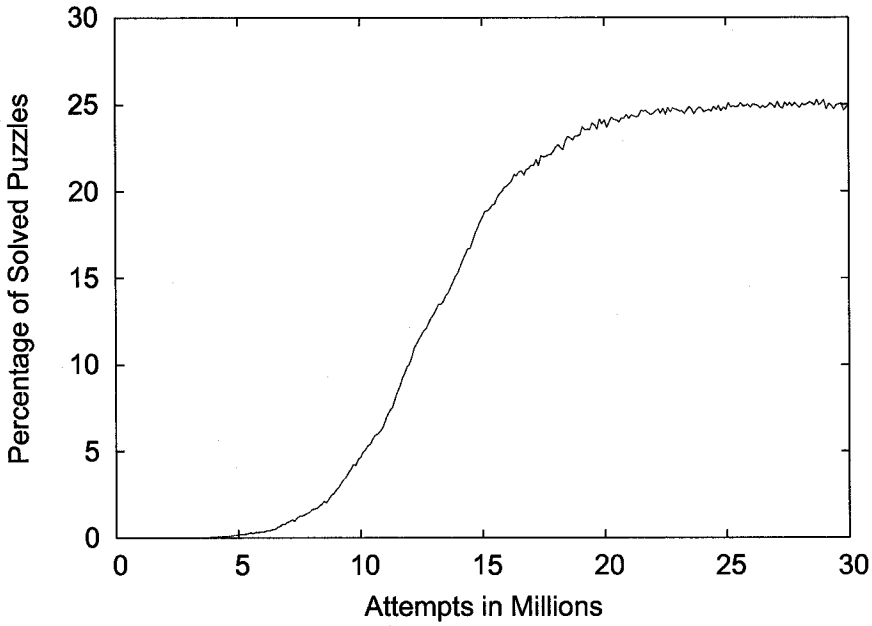

Figure 4.31: Solving torus 3×7 LightsOut as an n-ary goal using DQ-learning.

4.6.4.3 Solving 5×5 LightsOut using a 27-ary Goal

As we stated earlier, we were unable to solve a complete 5×5 puzzle as a binary goal due to memory requirements of over $2\frac{1}{2}$ gigabytes. Let us now calculate the amount of memory needed for the corresponding 27-ary goal. As before, the number of Q table entries can be obtained as $2^{25} \times 25 = 838,860,800$. Since we can still fit 27 different return values into the same 5 bits per table entry, the whole Q table can fit into $838,860,800 \times 5 \div 8 = 524,288,000$ bytes, or exactly "only" 500 megabytes! We were also able to simulate this learning task on the 1-gigabyte machines that we had access to.

Figure 4.32 shows the result of a typical experiment where after about 300 million learning trials, the puzzle was fully solved, reaching an approximately 25% solvability. We must point out that it took several days of machine time (process running in the background) for the DQ-

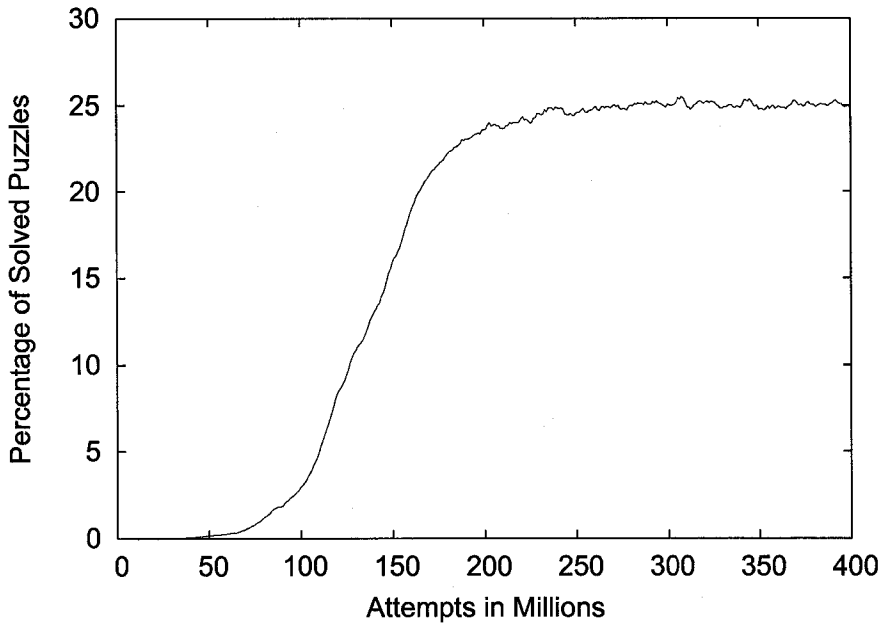

Figure 4.32: Solving classic 5×5 LightsOut as an n-ary goal using DQ-learning.

learning to converge, in this case. Is it really the case that only a quarter of the puzzles are solvable? The null space dimension for this variant is equal to 2 and, hence, the theoretical percentage of solvable patterns is $2^{-2} = 25\%$, fitting our experimental data almost exactly. In other words, only $2^{23} = 8,388,608$ puzzles out of a total $2^{25} = 33,554,432$ initial light patterns are solvable.

Figure 4.33 shows the result of a typical experiment with the "torus" variant, where after about 300 million learning trials, the puzzle was fully solved, reaching an approximately 0.39% solvability. We must point out that it took several days of machine time (process running in the background) for the DQ-learning to converge, in this case. Is it really the case that only such a small fraction of puzzles is solvable? We can observe that the fraction of solvable puzzles seems to be a power of 2, i.e. $2^{17}/2^{25} = 2^{-8} = 0.00390625 \approx 0.39\%$, fitting our experimental data almost exactly. In other words, only $2^{17} = 131,072$ puzzles out of a

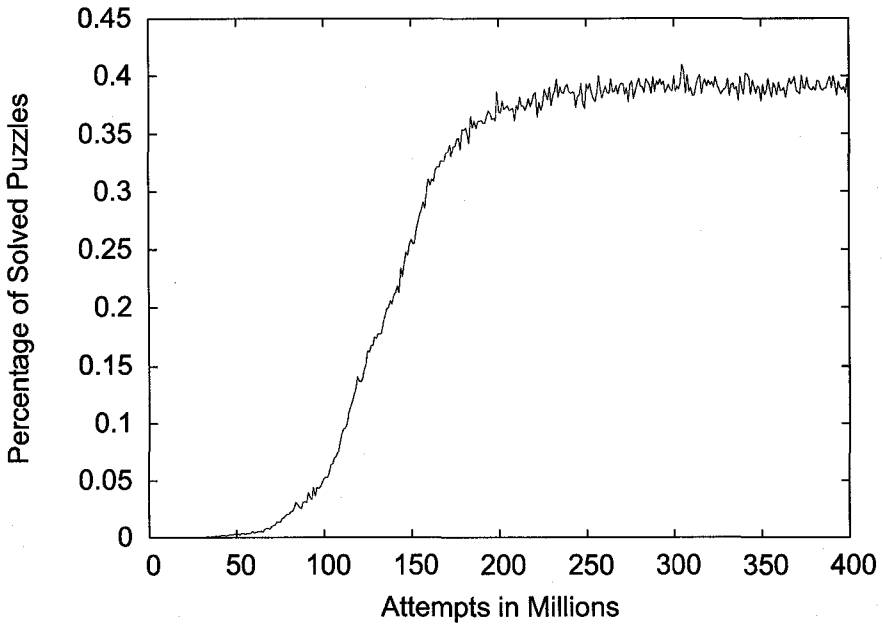

Figure 4.33: Solving torus 5×5 LightsOut as an n-ary goal using DQ-learning.

total $2^{25} = 33,554,432$ initial light patterns are solvable. Once again, this empirical result is easy to confirm by computing the pseudo-inverse matrix and determining the dimension of the null space. It is not a surprise then, that the dimension in this case works out to be exactly 8.

As with the 4×4 puzzle, we have also tried the Æip configuration with the Symmetry Map for this variant and the following Figure 4.34 shows one other curve resulting from learning trials where we take advantage of the symmetry of the puzzle. The curve shows the convergence of the learner when we take advantage of reflection, rotation, and "torus" shift symmetries. As a result, the convergence speed is dramatically improved by about 100 times! A single experiment no longer takes days of computing time.

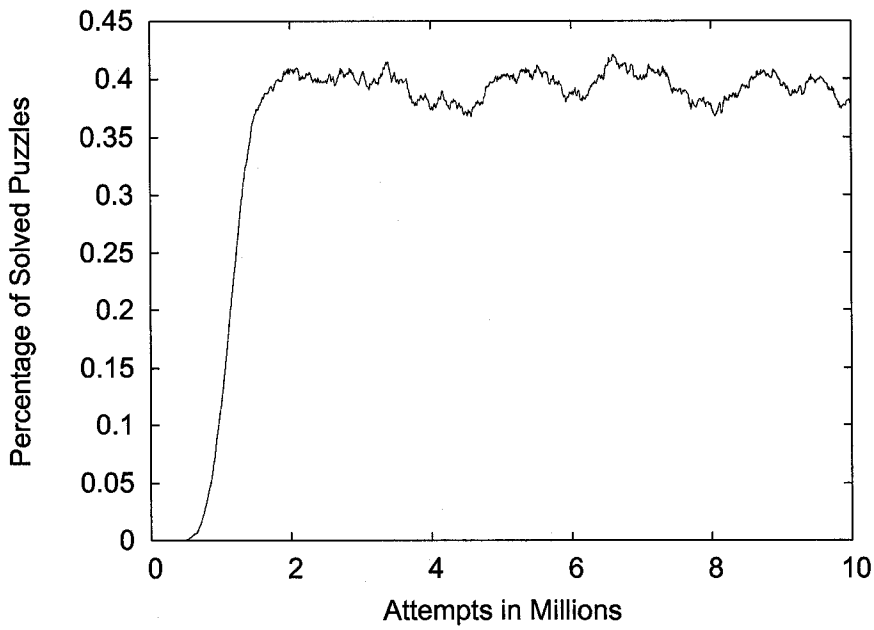

Figure 4.34: Solving torus 5×5 LightsOut as an n-ary goal using DQ-learning and taking advantage of reflection, rotation and torus shift symmetries.

4.6.5 Solving a "Difficult" LightsOut Variant

The essence of our approach is to demonstrate that a general method can learn a specific and difficult task of solving the LightsOut puzzle starting out with no prior knowledge of the rules of the puzzle. For example, initially the learner does not even know the association between selecting action i (interpreted by the LightsOut world as pushing on light i) and the effect this has on the corresponding light i, let alone its neighbours. The DQ-learner was able to solve the 5×5 variant of the puzzle *before* the author learned that an algebraic solution exists.

It would be interesting, however, to apply our method to a variant of this puzzle which cannot be analyzed in the same way. The algebraic analysis of LightsOut, presented earlier

and based on [8], [77] and [44], assumes that the effect of a pattern of presses on a pattern of lights can be represented as a modulo 2 sum of two binary vectors. It also assumes that a solution can be represented as a pattern of light presses, i.e. the shortest solution does not require any of the lights to be pressed more than once. We can, however, envision a simple modification of the puzzle for which these assumptions no longer hold. One possibility is to make the outcome of a light press *depend* on the current state of that light. For instance, when the light is off, pressing it would result in toggling a different neighbourhood of lights, e.g. the four diagonal neighbours (a "cross" instead of the "plus" neighbourhood). When the light is on, however, the effect is exactly the same as in the torus variant. We call this the "Difficult" LightsOut variant, because the previous algebraic solution does not apply — the effect of a light press can no longer be represented as a sum of two binary vectors.

We believe that it would be difficult, if not impossible, to find an analytical solution to this puzzle, and we are not aware of any attempts to find such solutions. We do not expect to find reported solutions of this variant as it does not appear among the many other variations of this puzzle already reported in the literature, such as the Lit-Only, Toggle, Orbix, XL-25, Merlin's Magic Square, LightsOut 2000, LightsOut Cube or LightsOut Deluxe, to name a few.

4.6.5.1 Solving the 5×5 Difficult LightsOut using a 27-ary Goal

We experimented with several grid sizes for the difficult variant of the puzzle. Here we describe the results of approaching the task of learning the 5×5 variant as a n-ary goal. Without careful analysis, we no longer have any means of estimating the upper bound on solution length, and so we had to make a decision to accept solutions of up to 25 steps in length just as we did with the torus 5×5 puzzle, still preferring shorter solutions. This decision is not as arbitrary as it may seem. In our experiments with smaller grid sizes the solutions lengths for the difficult puzzle did not exceed the upper bound that was established

for the regular puzzle. A successful analysis would confirm or disprove this property.

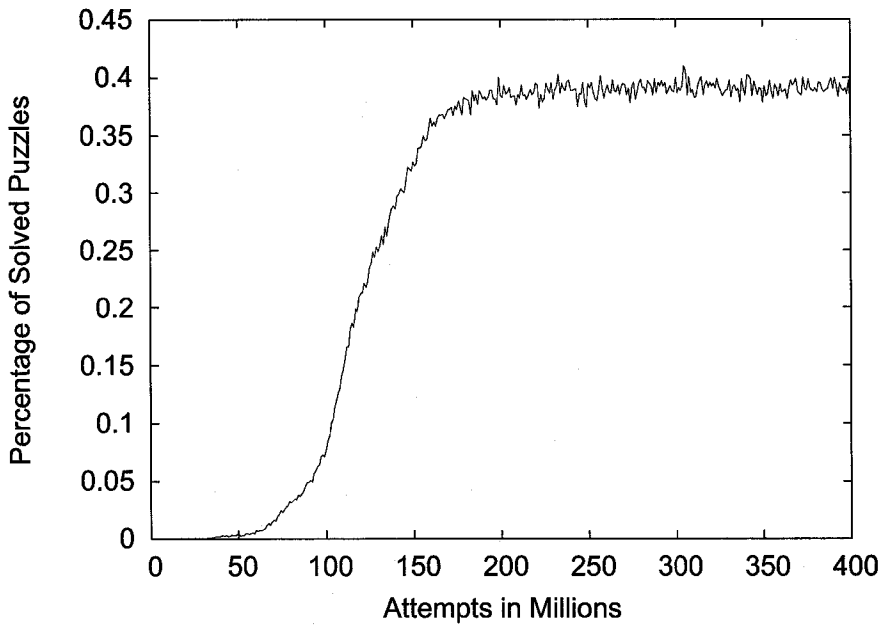

Figure 4.35: Solving the Difficult 5 × 5 LightsOut using DQ-learning.

Since the goal arity (27) is exactly the same as for the regular puzzle, the memory requirements for a DQ-learning algorithm would also be exactly the same, i.e. 500 megabytes. Figure 4.35 shows the result of a typical experiment where after about 300 million learning trials, a plateau of approximately 0.39% was once again reached. The plateau can be due to the fact that as for the regular 5 × 5 puzzle, only that fraction of puzzles is solvable. Another explanation would be that the other patterns are simply not solvable in only 25 light presses. In the absence of any mathematical analysis of the difficult puzzle variant, we can only increase the arity of the task and see if the fraction of solved patterns increases as well. But even in this case, we will never be sure (as we are lacking the upper bound on shortest solutions) whether such an increase is sufficient.

4.6.5.2 Comparison of LightsOut Experimental Results

The results of all of the reported LightsOut experiments are summarized in Table 4.1. It clearly demonstrates that DQ-learning and the related benefits of knowing the goal arity can make a difference between being and not being able to find a solution to a non-trivial problem given the practical memory constraints of today's computers. If more memory becomes available to our algorithm, we would be able to approach a LightsOut variant of a larger grid size, e.g. 4×7.

The DQ-learning algorithm applied to all the puzzle variants was not specifically adapted to solve LightsOut. Instead, in each of the experiments, it started out with virtually no knowledge of the rules of the puzzle or of what constitutes a successful solution to the puzzle. In three of the experiments, we were able to show that we can use the flexibility built into the Æip framework and provide information about the symmetries inherent in LightsOut using a separate Symmetry Map entity, i.e. without modifying the learning agent entity. Furthermore, we can see that by taking advantage of the symmetries of the puzzle we can dramatically reduce (by as much as a 100 times for a 5×5 torus variant) the time it takes to converge to a solution.

Summary

This chapter demonstrates how both single-player and two-player games can be modeled within the Æip framework using interconnected entities. In the context of Æip, we have set up several novel learning experiments where a standard Q-learning algorithm was shown to successfully learn a two-player game where virtually no initial knowledge of the game was provided, including even the basic fact that players must take turns. These experiments constitute the first contribution of this chapter.

We also proposed a DQ-learning algorithm as a discretized version of Q-learning based

Task	Arity	Conv'd Aft. Trials, 10^6	Q-lrng RAM, Mb	DQ-lrng RAM, Mb	Theor. Solvab'ty, %	Empirical Solvab'ty
4×4 classic	2	8	68	$2\frac{1}{8}$	6.25	match
4×4 torus	2	15	68	$2\frac{1}{8}$	100	match
3×7 classic	2	600	3686	$115\frac{1}{2}$	100	match
3×7 torus	2	600	3686	$115\frac{1}{2}$	25	match
4×4 classic	18	0.5	4	0.625	6.25	match
4×4 torus	18	1.5	4	0.625	100	match
4×4 torus sym	18	0.2	4	0.625	100	match
4×4 torus sh.sym	18	0.05	4	0.625	100	match
3×7 classic	23	60	168	$26\frac{1}{4}$	100	match
3×7 torus	23	30	168	$26\frac{1}{4}$	25	match
5×5 classic	27	300	3200	500	25	match
5×5 torus	27	300	3200	500	0.390625	match
5×5 torus sh.sym	27	3	3200	500	0.390625	match
5×5 difficult	27	300	3200	500	0.390625	N/A

Table 4.1: Summary of experimental results for LightsOut. Note that there is no known analytic or search-based solution for the difficult variant of the puzzle. Also note that in most cases, the Æip implementation is not given any notion of the rules of the puzzle — not even the concept of which are the neighbouring lights for a given light being manipulated.

on the feedback signal proposed in Chapter 3. We showed the advantages of this algorithm in terms of significant memory savings given the knowledge of goal arity. This is one of the main contributions of this chapter. This, in turn, led to an unexpected discovery that, given the goal arity, the same number of bits per table entry seem to be sufficient regardless of whether the agent entity is facing a deterministic or a non-deterministic environment. Moreover, the fact that the feedback signal is, by definition, monotonically decreasing, implies that for a non-deterministic environment, the table entries should also change in only one direction, which allows the DQ learner to make fewer updates and results in a potentially faster convergence. This discovery is another contribution of the chapter.

Through the reported experiments we, first, confirmed the memory savings in the simple case of tic-tac-toe, and then successfully approached the much more difficult problem of learning to solve the LightsOut puzzle. In the latter case, we would not have been able to solve the puzzle using the standard Q-learning implementation due to prohibitive memory requirements. In other words, the knowledge of goal arity can make a difference between being and not being able to find a solution to a non-trivial problem given the practical memory constraints of today's computers. Finally, we showed how our methodology can be used to find a solution to a difficult variant of the puzzle, for which no known analytical or any other solutions existed before.

Chapter 5

Comparing List Organizing Strategies with Æip

Introduction

The previous Chapter demonstrated how the Æip framework could be used to setup learning experiments in the game playing domain, where a superior player is determined through a competitive play. We believe we can use the same Æip framework and extend this idea to a broader domain of realistically comparing "algorithms", whereby a superior (or more robust) algorithm that achieves a specific task can be determined through "competition". Since our primary interest is in comparing learning algorithms, we now turn our attention to the domain of comparing adaptive data structures and, more specifically, adaptive list organizing strategies.

In this Chapter we consider the problem of comparing the performance of well-known adaptive algorithms that organize a linked list, and maintain it in an approximately optimal order with respect to the mean search time. The traditional setting for adaptive list organization assumes that information is stored in the form of key-value pairs as elements of a

singly linked list. In order to lookup the information stored in any specific list element, the list must be traversed starting from the front element until the required element is found, or the end of the list is reached. If such searches are conducted often, it pays to optimize the order in which the list elements are stored. Ideally, the elements should be arranged in the descending order of the respective access probabilities. Additionally, although the exact probabilities are not known in most practical applications, we assume that they are not estimated based on the history of past requests, but that rather the list organization is achieved without such an explicit estimation phase.

In order to estimate the access probability for an element as precisely as possible, we need to maintain a frequency count for every element of the list. If the list has a significant number of elements (e.g. in the thousands), additional memory requirements for such an estimation will also be significant. Surprisingly, we can organize the list very well without such an explicit frequency count. This is achieved by making incremental adjustments to the order of the list after each request for an element. This principle forms the basis for the adaptive list organizing algorithms such as the Move-To-Front, the Transposition, and several of their derivatives.

When compared individually, the Move-To-Front (MTF) strategy converges faster than the Transposition (TR) strategy, while the latter approaches optimality closer over time and, thus, can be labeled as a more accurate algorithm. Consequently, when accuracy is more important than speed, the TR strategy should be chosen. When speed is of overriding concern, the MTF strategy is preferable. If, however, we utilized a metric, which combined both speed and accuracy, the question of determining a superior scheme is of particular interest. To date, there is no such universal metric — indeed, how could we weigh the speed and accuracy in a single measure? If we are able, however, to make these (and, in general, any other) schemes play a conceptual competitive game against each other, such a comparison would be possible. This is our present endeavour.

In this Chapter, we propose to use the Æip framework to compare these and other

algorithms in a novel setting of competitive list organization. We do this by requiring that two algorithms attempt to sort the list in *opposite* directions[1]. The unexpected result of this study is that algorithms that are considered superior based on individual merits, often fare worse in a competitive setting, and vice versa. We feel that such a competitive setting is appropriate and promising for the empirical comparison of algorithms in various application domains, including problems requiring "intelligent" algorithms. To present our contribution in the right setting, we first include a brief history of this field as well as *some* of the known results, and then describe our novel experiments and their outcome.

5.1 Adaptive List Organizing Algorithms

In this Section we provide the essential concepts pertaining to Adaptive List Organizing strategies. Our account closely follows a recent review performed by Abdelrahman Amer [6]. Additional details based on the same review are included in Appendix B. The general field of Adaptive Data Structures involves a study of self-organizing data structures which includes linear structures (e.g. linked lists) as well as two-dimensional structures (e.g. trees). In this Chapter, we are not going to concern ourselves with the latter, and will concentrate only on linked lists and the corresponding algorithms. Furthermore, we will make an assumption that each data access is independent of all the past accesses. We will also restrict our review to deterministic algorithms.

5.1.1 Self-Organizing Lists

A *self-organizing data structure* is a data structure that employs a rule (often referred to as a scheme or strategy), for changing the arrangement of its elements in order to minimize

[1]We are not aware of any reported research which compares list organizing strategies in such a "competitive" fashion.

some notion of cost, e.g. the average access time. The rule is commonly applied after every access to the list.

The access probabilities are usually assumed to follow some distribution, unknown to the self-organizing data structure. For the purposes of analyzing the performance of the data structure, we further assume that records not found in the list will never be requested, and that each record will be accessed at least once [33].

Suppose we have a *list* of records $R_1, R_2, \ldots, R_n$ in an arbitrary order π, where R_i is in position $\pi(i)$ for $1 \leq i \leq n$. The record R_i is accessed with unknown probability, s_i. When R_i needs to be accessed, a sequential search is performed starting with the first record, the operation costing $\pi(i)$ units of time. The expected search cost for π would then be:

$$cost(\pi) = \sum_{1 \leq i \leq n} s_i \pi(i). \tag{5.1}$$

The goal of the data structure is to minimize the search cost. This can be achieved if the records of the list are arranged in an optimal manner, that is, they must be arranged in the descending order of their access probabilities.

In the literature this problem can be referred to by a variety of names, including online list organization, the adaptive list update problem, self-organizing linear search, or simply, the study of self-organizing lists. In this Chapter we will often refer to such lists by a more general term — adaptive data structures.

A *self-organizing list* utilizes a strategy for altering the position of its records, often with every access, so as to approach the optimal ordering where the most frequently accessed records are closer to the front of the list.

While in this Chapter we are going to restrict our attention to lists, we must point out that there are other extensively studied self-organizing data structures, such as trees. While more difficult to implement, these structures can reduce the search time considerably. List data structures are still, however, very popular both in research and practice, because of their simplicity and ease of implementation. Another reason why our study concentrates on

the lists, is that it is quite natural to set up competitive experiments of the kind we are reporting in Section 5.4.

5.1.1.1 Cost

Since the self-organizing data structures are tasked with reducing the access cost, it is important to define what we mean by cost, especially since the term "cost" can mean many different things.

The *total time* cost associated with accessing a record in a data structure is the sum of the cost required to find the record, and the cost involved in reordering the list after applying the permutation algorithm. Interestingly, the reordering cost can often be a constant, or quite small when compared to the search cost, and is therefore negligible or ignored. A cost model where the reorganization cost is not considered, is called *free exchange*. In contrast, the *paid exchange* cost model (P^d) is where the cost of reorganization alone involves d units [3]. Most of the literature, however, assumes the standard cost model, and we are going to do likewise.

The general notion of cost discussed above can be further decomposed into analysis of asymptotic cost, amortized cost, and relative cost discussed in Appendix B.

Due to sometimes insurmountable difficulties in analytical comparison algorithms, researchers often resort to comparing algorithms empirically, where a simulation of the studied data structure is implemented, and the average asymptotic costs are calculated. By taking ensemble averages over all the experiments performed, as well as the time average within a single experiment, the final convergence statistics can be computed. For ergodic systems, ensemble and time average are equivalent. These averages "smoothen" the curve of the observed cost, and an accurate estimation of the asymptotic cost can be obtained. We also take an approach of empirical comparison in the reported results in this Chapter. Since we generated query distributions at random instead of choosing one particular form of a distri-

bution (see Section 5.2 for the description of our simulation methodology), we also needed to average the performance of our algorithms over a set of different equiprobable distributions.

5.1.2 Basic Schemes

The simplest way to organize a list in the descending order of access probabilities is employed by the *frequency count* (FC) algorithm, which keeps a counter for each record, and, thus, estimates the access probabilities by using observed access frequencies. Whenever a record is accessed, its corresponding counter is incremented, and the record is then moved forward in front of all the records with lower access counts.

The FC algorithm is simple and it yields some impressive analysis results: its asymptotic cost is the same as that of the optimal static ordering, and its amortized cost is at most twice that of the optimal [17]. This algorithm, however, has obvious drawbacks. First of all, it is a linear-space algorithm, which requires as many counters as there are records. Remarkably, there exist algorithms that need only constant memory, and that is the reason why FC is not attractive from the point of view of memory requirements. Besides the linear memory requirement, FC is relatively insensitive to locality of reference [33, 79], meaning that it would perform poorly if there is dependence among the access requests. Furthermore, it appears that FC requires an extra non-constant search to find the new position where the accessed record has to be placed. Finally, FC is not c-competitive for any c [79].

Apart from FC, the two most famous constant-space, self-organizing schemes for linear lists are the *move-to-front* (MTF) and *transposition* (TR) rules, introduced by McCabe [45]. Both schemes change the position of the accessed record. In MTF, a record is moved to the front of the list, unless it is already the first record. In TR, the record is moved one position closer to the front of the list by interchanging it with its preceding record unless it is at the front of the list. Most of the other deterministic schemes can be seen to be variations of these two. See Appendix B for analysis of MTF and TR.

5.1.3 Combining MTF and TR Schemes

The MTF and TR rules have their own strengths and weaknesses. MTF is superior when quick convergence and a quick response to a switching environment is required. It also performs better in environments characterized with locality of reference [22]. TR excels if stability is desired and quick convergence is not essential. Comparing these in a fair competitive setting is the goal of this Chapter. Researchers have suggested several ways to combine these algorithms to benefit from their respective advantages.

5.1.3.1 Bitner's Algorithm

Bitner [20] proposed an algorithm which behaves initially as the move-to-front rule to exploit its quick convergence property, then switch to transposition to benefit from the lower asymptotic cost and higher stability. Determining when to switch, however, is not straightforward and is purely assumptive. In the remainder of this Chapter, Bitner's algorithm will be referred to as MTF⇒TR so as to maintain a consistent naming convention. It is used in a number of reported experiments.

5.1.3.2 Move-ahead-k (MHD(k))

Rivest [68] proposed the move-ahead-k rule as a compromise between MTF and TR. In move-ahead-k, when an element at position j is accessed, it is moved k-positions forward to the front of the list unless it is in the first k positions, in which case it is moved to the front. Observe that move-ahead-1 is the same as the transposition rule, while move-ahead-n is the same as move-to-front. As such, the performance of move-ahead-k is expected to lie between the two. However, there is no formula for the average search cost of move-ahead-k as its analysis is difficult. Tenenbaum [82] carried out simulations on the move-ahead-k rule that showed that the best value of k is correlated to the list size.

Our own experiments did not involve this hybrid algorithm. In addition to the plain TR

and MTF, we chose to compare the MTF⇒TR and MTF⇔TR variants which are essentially interleavings of the plain TR and MTF rules. These schemes are described in detail later. At the end of this Chapter we also introduce another hybrid scheme (termed RTF+TR) purposefully designed to take advantage of the knowledge of the opponent algorithm.

5.2 Simulating Queries with Equiprobable Distributions

The relative effectiveness of a particular list organizing strategy often depends on the underlying query probability distribution. A query probability distribution is an assignment of probabilities of requesting each of the list elements. Obtaining and utilizing such a distribution is fundamental to the issue of simulating and testing the various schemes in question. With the aim of simplifying the nomenclature of this section (and in order to not confuse the reader), we shall refer to the query probability distributions as *query patterns.*

In order to empirically assess the performance of algorithms in either the traditional or the competitive settings (both settings are described later), we must be able to demonstrate that similar results are obtained not only for one specific pattern, but also for a variety of different query patterns. Typically, this is accomplished by selecting a handful of well-known parametric distributions, such as the earlier mentioned Zipf, 80-20, Lotka, exponential, linear, etc., and then demonstrating that the desired properties of an algorithm apparently hold for each of these query patterns. Sometimes the choice of a specific pattern can be justified based on the knowledge of the application domain. In the case of adaptive data structures, however, the whole point of using adaptive algorithms is precisely because we don't know the underlying pattern, and that it often cannot be described in a parametric form. So, for our purposes of testing the algorithms, resorting to a particular choice of query patterns would be rather arbitrary.

We believe that we can remove the issue of arbitrariness in choosing a suitable query pattern and simply opt to generate a random representative sample of *random* query patterns.

The intention is that each pattern in such a sample is just as likely to be selected for testing as any other possible pattern over a given size of the list. In other words, we would like to guarantee that the probabilities of selecting any of the infinitely many different query patterns are themselves distributed uniformly. We will refer to query patterns selected in this fashion as *equiprobable patterns.* The larger the size of the sample of such equiprobable patterns used for testing our algorithms, the more certain we would be of the conclusions drawn from the empirical simulation, simply because we would be more likely to test against a wide variety of qualitatively different query patterns.

How can we generate a pattern (query distribution) which is chosen in a uniform manner from the set of all possible patterns? In other words, how can we generate an equiprobable pattern? This question seems to be simpler than it actually is. For a more detailed discussion of the generating procedure the reader is referred to Appendix C.

Suppose we want to generate a query pattern of size n, i.e. a probability distribution over n queries. The essence of the procedure is to generate $n-1$ random numbers

$$\{u_i\} \quad i = 1, 2, \ldots, n-1,$$

sort them in an ascending order to obtain $\{u_{s_j}\}$, where $\{s_j\}, j = 1, 2, \ldots, n-1$ are the indices i in the sorted order, and then assign

$$\begin{aligned} p_1 &= u_{s_1} \\ p_k &= u_{s_k} - u_{s_{k-1}} \quad k = 2, 3, ..., n-1 \\ p_n &= 1 - u_{s_{n-1}}. \end{aligned}$$

This is precisely the algorithm we used in all our experiments. While the sorting step adds extra complexity to the algorithm, we feel it is worth the extra effort in order to have higher empirical confidence in the performance of an algorithm.

5.3 Comparison of Algorithms in the Traditional Setting

By the traditional setting, we mean the comparison method where two algorithms are organizing individual lists in two separate experiments, and where their convergence and accuracy is plotted side by side to determine the superior one given some chosen criterion (such as the best accuracy). We call it the *traditional* setting in order to contrast it with the *competitive* setting described in Section 5.4. Except for confirming the already known properties of tested schemes when simulated on equiprobable query patterns, this section does not include new results and is included solely for comparison with the results obtained in the competitive setting.

5.3.1 Æip Configuration

In order to setup up such traditional experiments within the Æip framework, we first need to decide what entities are required. In this case, it is natural to create the ListWorld entity which represents the list data-structure and the Scheme entity (e.g. TR entity) that encapsulates the organizing strategy. There are several possible ways by which these two entities can interact with each other — much depends on the exact information exchanged.

In the experiments described here, we decided to implement a fairly high level of interaction between the entities, whereby the Scheme entity emits the list reorganization operator to be applied to the list after the next query, and the ListWorld responds with the query cost resulting from the last query. For simple schemes like TR or MTF the operator that their corresponding entities emit would always be the same, whereas for a combined scheme such as MTF$\Rightarrow$TR or MTF$\Leftrightarrow$TR[2], the operators would change from time to time.

To be precise, the ListWorld entity has a single outport with the corresponding signal

[2]These schemes are introduced in Sections 5.3.3 and 5.4.6.

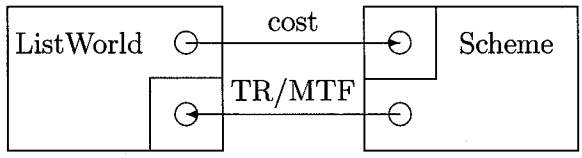

Figure 5.1: Æip configuration for traditional list organizing experiments.

in most practical applications

being the cost (a non-negative integer) of the last query. This signal is fed into the sole inport of the Scheme entity. The only outport of the latter, on the other hand, carries the list reorganization operator that is accepted by the only inport of the ListWorld. Upon receiving the operator (implemented as a Java object), the ListWorld applies this operator to the current order of the elements. We must emphasize that in the chosen Æip configuration, the Scheme entity receives little or no information regarding the *location of the elements* within the list. While the cost of the last query (which is the only information that the Scheme entity receives) is essentially the location of the last queried element, the Scheme entity *does not know* exactly which element was queried. The actual reorganization of the list is performed by the ListWorld itself upon reception of the operator from the Scheme entity. Since the list knows which element was accessed, it can easily apply operators such as transposition or move-to-front.

What follows is a description of comparisons of three simple strategies within the Æip framework according to the configuration presented. These comparisons are included only in order to contrast them with comparisons in the competitive setting.

5.3.2 Transposition and Move-To-Front

As explained earlier, the Move-To-Front (MTF) algorithm, after having found the newly requested element, always moves it to the front of the list. The Transposition (TR) strategy, on the other hand, transposes the found element with the element in front of it. In other

words, it moves the requested element towards the front of the list by one position. Both of these simple strategies do nothing if the requested element is already the first element of the list. For further description and analysis of these simple strategies, we refer the reader to the earlier Section 5.1.

To perform an empirical comparison of TR and MTF, we have created a list with $N = 100$ elements and tested each strategy on a 100 equiprobable query patterns. We ran 100 simulation runs for each of the patterns in the sample. Each run consisted of 5,000 simulated queries.

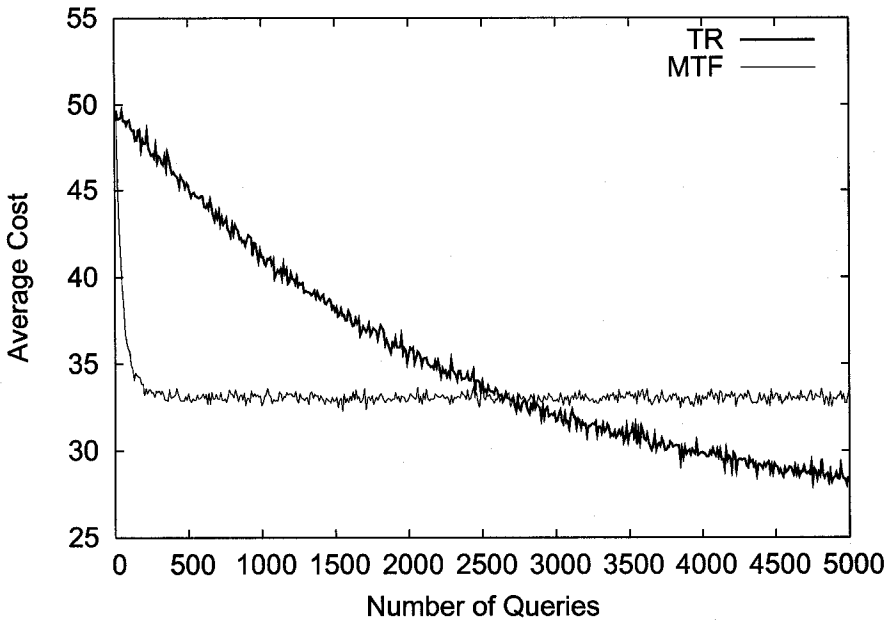

Figure 5.2: TR and MTF in the traditional setting. Over time TR approaches the optimal cost (≈ 25) closer than MTF.

Figure 5.2 shows a comparative plot of the two strategies averaged over 10,000 simulation runs (100 simulation runs for each of the 100 patterns) with 5,000 queries per each simulation. This plot confirms the well-known results that MTF converges faster, but to a less optimal

cost ($\approx$ 33, in this case), while TR takes its time to reach closer to the optimal cost, i.e. $\approx$ 28 as opposed to the optimal $\approx$ 25.[3] The TR will, in fact, converge to a value close to the optimum, and it is simply not shown in the figure as the experiment was stopped prior to attaining this.

5.3.3 TR and MTF⇒TR

Can we, perhaps, combine the advantages of both basic schemes into a single scheme? We can start out with the MTF scheme, let it quickly rearrange the list and as soon as MTF reaches its plateau, we can switch to the TR scheme to ensure convergence to the optimal cost. In order to decide when to switch to TR, we can gauge the running average cost at certain intervals and check to see if the cost stopped decreasing. As soon as the last gauged average cost increases, we switch the scheme. The extra memory required for such a combined scheme is independent of the length of the list, i.e. the scheme is constant-space, and therefore does not increase the complexity of the algorithm. This strategy was first published by Bitner [20] and referred to as Bitner's algorithm in our review, but we will refer to it as MTF⇒TR for added clarity.

Figure 5.3 shows a comparative plot of the two strategies averaged over 10,000 simulation runs (100 simulations runs for each of the 100 patterns) with 5,000 queries per each simulation. The *gauging interval* for the plot in the figure was set to 100 queries. In other words, the algorithm was determining whether it was time to switch to a TR scheme on every 100^{th} query, by comparing the current average accuracy with the one recorded a 100 queries earlier.

---

[3]The optimal cost was computed for each equiprobably generated pattern and then averaged over the 100 patterns used. It is worth noting here that the average optimal cost for equiprobable distributions turns out to be always equal to one quarter of the length of the list. This is a phenomenon yet to be confirmed theoretically.

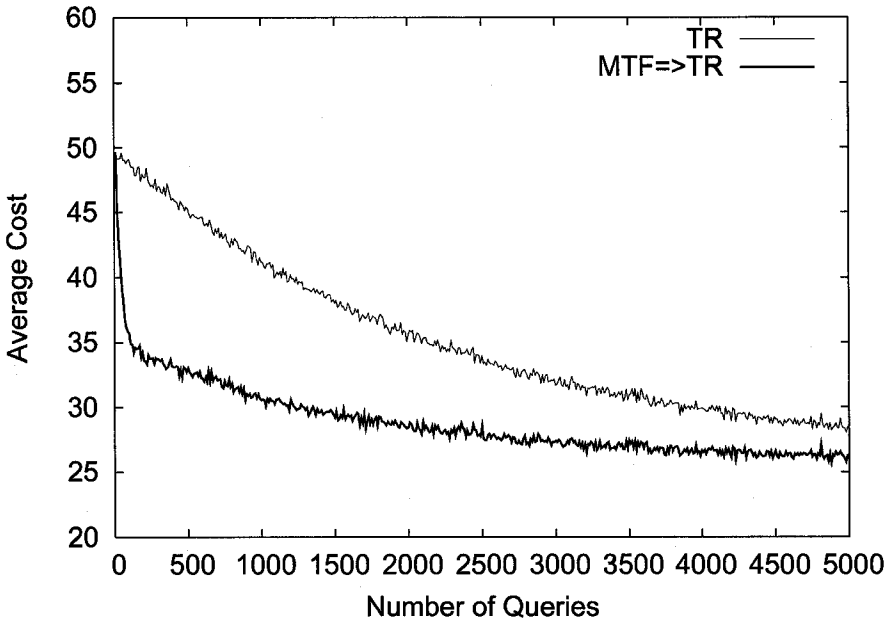

Figure 5.3: TR and MTF⇒TR in the traditional setting. Fast initial convergence of MTF is replaced by a more accurate convergence of TR. Over time the combined scheme approaches the optimal cost of ≈ 25 just as the ordinary TR, but much faster.

Informally explained, we essentially take the best from both schemes and produce a winning combination. We can observe two disadvantages of MTF⇒TR. First, it now has a parameter — the gauging interval — with the following properties: the smaller the interval, the more likely the algorithm will switch to TR prematurely (i.e. due to a random fluctuation in cost); whereas the larger the interval, the longer it will take to switch to TR and, hence, converge. At present it is not clear how one can set this parameter in any given application. Second, should the underlying query pattern suddenly change, or, equivalently, (as we will see in Section 5.4.6) should the arrangement of elements be disturbed by a third party with access to the same list *after* the switch to TR had occurred, the scheme would essentially remain TR, losing the benefits of the fast initial convergence of MTF.

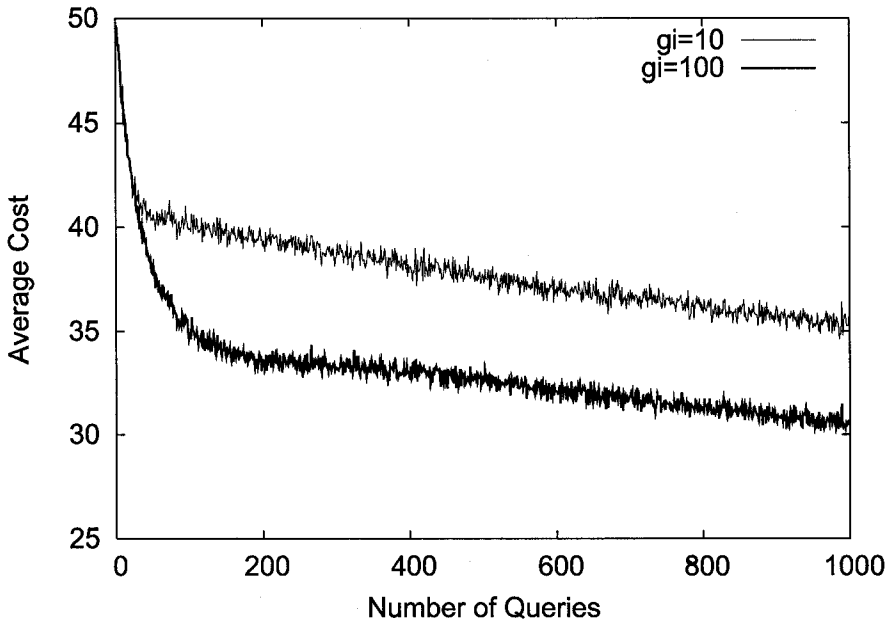

Figure 5.4: MTF⇒TR for two gauging intervals: 10 and 100. With the gauging interval of 10 the scheme switches to TR prematurely.

To see the effect of choosing the gauging interval, Figure 5.4 compares, in a side-by-side manner, the convergence of MTF⇒TR for gauging intervals of 10 and 100. In our experiments, the MTF scheme alone would reach the plateau after about a 100 queries, which suggests that a good heuristic may be to set the gauging interval to be equal to the size of the list. Whether this heuristic will work well for lists of significantly different sizes or for different query distributions, remains to be shown empirically. One can clearly see from the figure, that with the gauging interval of 10, the scheme switches to TR prematurely.

5.4 Comparison in the Competitive Setting

We can think of a number of ways by which different list-organization strategies can be made to compete against each other. For example, we can assign a different query source (and, therefore, a different query pattern) to each competing scheme, and alternate queries between the two schemes. The scheme that converges to the lowest average access cost would be declared a more robust scheme. In the experiments described below, however, we chose a different competitive setting. Both schemes process queries of the same pattern (i.e. generated by the same underlying distribution), but each scheme searches the element from the opposing end of the list. Both schemes alternate queries as before. To put it simply, the competing schemes are organizing the list in opposite orders. Once again, the more robust algorithm would reach the lower average access cost over time.

In order to setup such a competition, we need to transform our list data structure into a doubly-linked list, so that both move-to-front and transposition operations are constant-time from both ends of the list.

5.4.1 Æip Configuration

The Æip configuration for such a competitive setting must be modified to accommodate a new Scheme entity. Instead of having only one inport and one outport, the ListWorld now has two inports, which accept operators from the two Scheme entities, and the two outports which emit the respective costs of each query. Except for experiments with enabled *balancing* (see Section 5.4.3), the ListWorld would generate a new query based on the current distribution and alternate in accepting the reorganization operator between the two Scheme entities. The ListWorld entity assigns a different end of the list as the virtual front (of the list) to each of the two Scheme entities. We must emphasize that in such a configuration, neither Scheme entity is aware of the opponent and assumes that the list is a singly-linked structure, with the understanding that external forces might be perturbing the order that

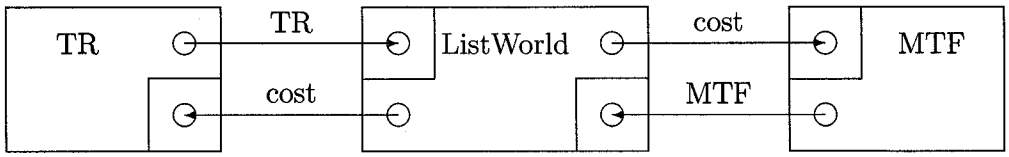

Figure 5.5: Æip configuration for competitive list organizing experiments.

either Scheme entity is trying to achieve.

As an example, Figure 5.5 shows TR and MTF schemes competing within the same ListWorld environment. In this Section, we also comparatively consider variations of MTF, namely, MTF⇒TR and MTF⇔TR described below.

5.4.2 TR versus MTF

To perform an empirical comparison of TR and MTF in the competitive setting, we have created a doubly linked list with a 100 elements and ran a 100 simulations of the two algorithms competing against each other for each of a 100 randomly selected equiprobable query patterns. Each simulation run consisted of 1,000 queries.

Figure 5.6 shows a comparative plot of the two strategies. It is clear from the plot that TR offers no competition to the agile MTF which converges to virtually the same cost (i.e. $\approx$ 34) as in the traditional setting. TR, on the other hand, is not too far from the worst expected cost of $\approx$ 75, when the list is in the exact reverse to the optimal order. At first, this seems rather surprising as TR is a definitive winner in terms of accuracy in the traditional setting. We observe, however, that the two algorithms were taking strict turns at rearranging the list, and each move-to-front operation of MTF is a much more destructive operation to the TR order, than a single transposition is to the MTF order. This leads us to consider a way of *balancing* the two algorithms in the next section.

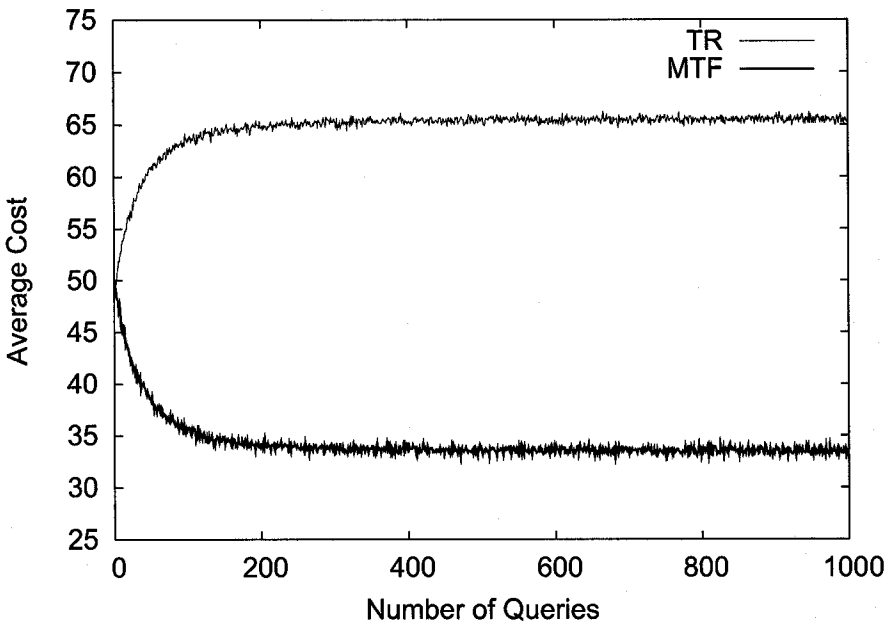

Figure 5.6: TR versus MTF in the competitive setting. The MTF strategy is preventing the TR strategy to organize the list.

5.4.3 TR versus MTF in a "Fair" Competition

Is the fact that MTF significantly outperforms TR in a competitive setting, due only to the difference in the mutual destructiveness of the respective operators? We can verify this by trying to level the playing field and set up a more "fair" competition where we attempt to balance the TR and MTF operators.

We can view a single move-to-front operation as a combination of multiple transposition operations. For example, if the element being accessed is in position with index 3 (indices start at 0), the query cost is also 3 (three node traversals). This element can be moved to the front of the list with either a single move-to-front operation, or equivalently, with 3 consecutive transposition operations, where the element in question is being transposed with

the element immediately in front of it.

Based on this observation, we can balance the number of transpositions performed by either opponent. Instead of always alternating between the competing schemes, we can instruct the ListWorld to allow an MTF operator to proceed only after an equivalent number of the TR operators were performed by the opponent. In other words, in a TR versus MTF competition, the TR scheme is usually allowed to reorganize the list multiple times before the turn is passed to the MTF scheme.

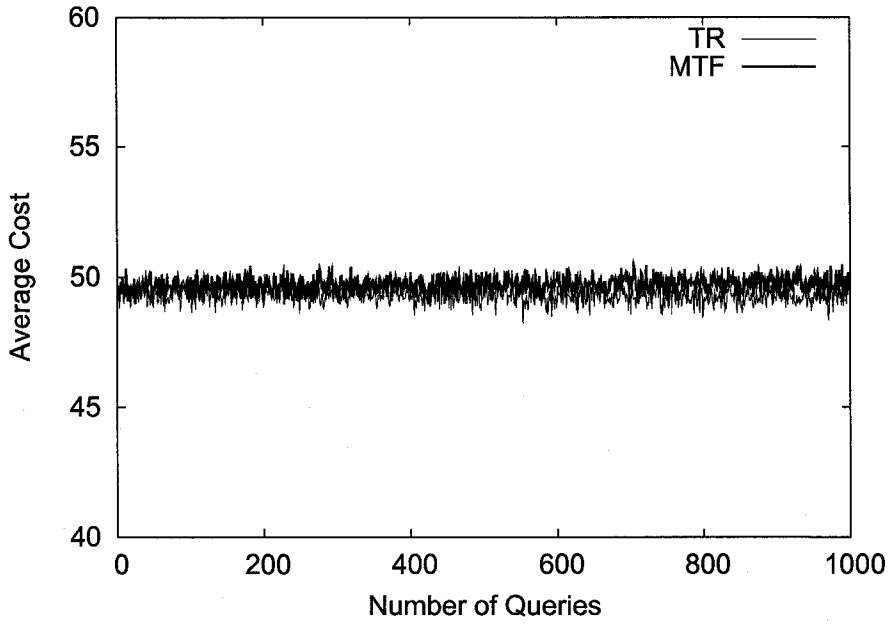

Figure 5.7: TR versus MTF in a balanced competition. Both schemes fare equally in a balanced competition.

Figure 5.7 shows the result of an experiment where the same 100-element doubly-linked list was used in 10,000 simulations (100 runs for each of the 100 random query patterns) with exactly 1,000 queries per each simulation. We can see no significant difference in the performance of the two schemes, which suggests that the differences seen in the unbalanced

competition are solely due to the relative destructiveness of TR and MTF operators. We must note, however, that in a practical application, it would be unreasonable to assume the knowledge of the external destructive force acting on the list and, hence, it would be impossible to balance the reorganization operations. The main purpose, therefore, of the balanced experiment is to empirically confirm the reasons behind MTF's success in the competitive setting.

5.4.4 MTF versus MTF⇒TR

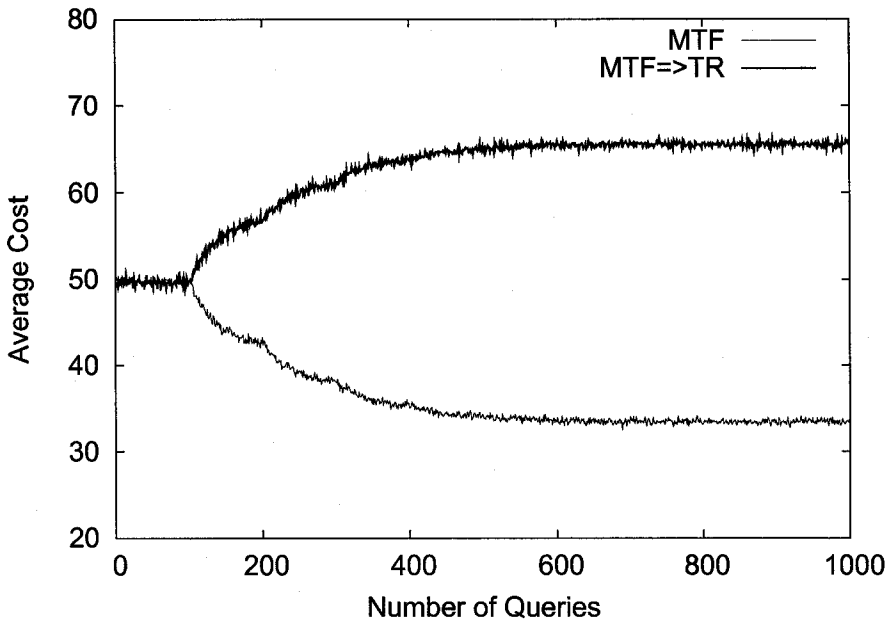

Figure 5.8: MTF versus MTF⇒TR in a competitive setting. After switching to TR, MTF⇒TR is quickly outperformed by MTF.

We have seen earlier in Section 5.3.3, that in the traditional setting, MTF⇒TR scheme offers clear advantages. We also made a remark that this scheme has a potential disadvantage

when faced with an external "order destroying force", such as the opponent in our competitive setting. If the opponent is a plain MTF scheme, after MTF⇒TR makes a permanent switch to a TR scheme, the competition would be between a plain MTF and a plain TR. We know that in such a competition the MTF would be a clear winner. We can now confirm this conclusion empirically.

Figure 5.8 shows the results of a competition between MTF and MTF⇒TR averaged over 10,000 simulation runs (100 runs for each of the 100 random query patterns, as before). The gauging interval for MTF⇒TR was set to 100, and the scheme switched immediately after the 100 queries because no increase in accuracy could be achieved. Both schemes initially start out as MTF and neither can gain an advantage over the other until a switch to TR occurs in one of them. Soon after, the plain MTF regains the competitive edge becoming a clear winner, and the final plot is identical to that of Figure 5.6, where MTF reaches a cost of ≈ 34, while that of MTF⇒TR increases to ≈ 66. Thus, we have empirically confirmed that MTF⇒TR is a poor scheme to use in a setting where an external destructive force is applied to the list.

5.4.5 TR versus MTF⇒TR

Since MTF⇒TR is eventually going to act as a plain TR scheme, how would it fare against another plain TR scheme? It would be quite logical to expect both of them to perform equally soon after a switch to TR occurs in MTF⇒TR. Figure 5.9 shows the results of a competition between TR and MTF⇒TR under the same conditions as in the previous section. The gauging interval was once again set to 100.

The plot seems rather surprising as we expected the two schemes to converge soon after a switch to TR by MTF⇒TR. Is there a mistake in the simulation? There is a good lesson here in drawing quick conclusions from empirical data. If the simulation is extended for many more than 1,000 queries, we would, in fact, witness the convergence of the two schemes.

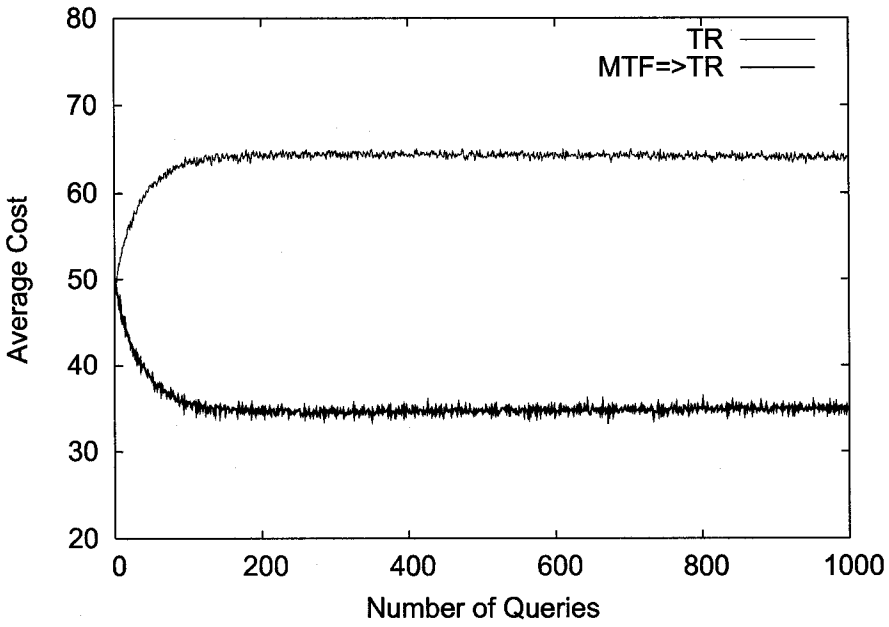

Figure 5.9: TR versus MTF⇒TR in a competitive setting. MTF⇒TR retains its advantage long after a switch to TR.

Figure 5.10 shows what happens if the experiment is extended to over 100,000 queries. The horizontal axis has a logarithmic scale to contrast the initial divergence with the ultimate convergence of the two schemes. This result only underscores the extremely slow nature of TR convergence. If the list was longer than 100 elements, we would expect a much longer period of time during which MTF⇒TR still retains its advantage over a plain TR after the switch to a TR scheme had occurred.

5.4.6 MTF versus MTF⇔TR

We have seen in Section 5.4.4 that MTF⇒TR is not a good scheme to use in an environment with external interference to the list order. We can clearly attribute this conclusion to the

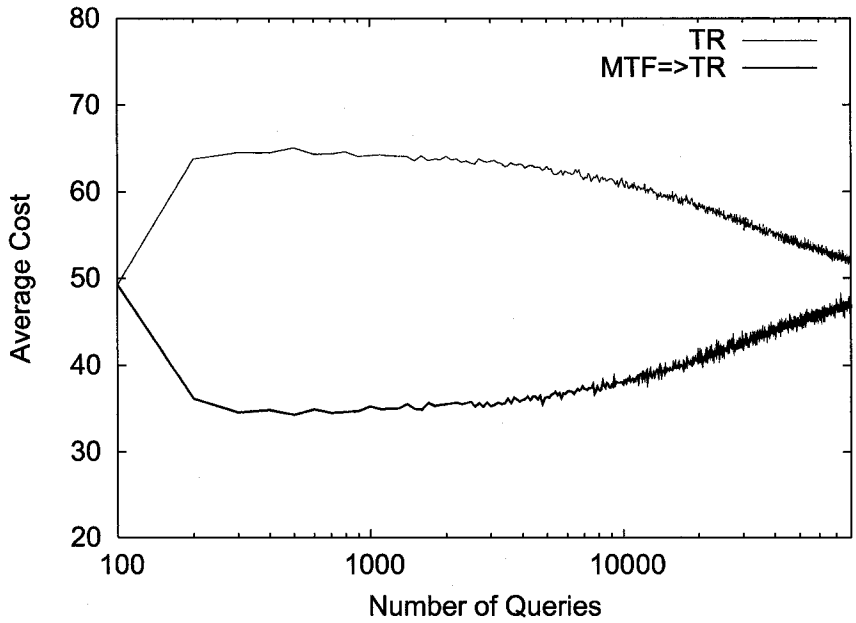

Figure 5.10: TR versus MTF⇒TR in a competitive setting. Eventually MTF⇒TR and TR converge. Observe the logarithmic scale for the "Number of Queries" axis.

fact that a permanent switch to TR scheme must occur. It would be natural to modify this algorithm so that the schemes can be switched back and forth between MTF and TR depending on which one seems to offer a better prospect of lowering the average cost. This strategy would work well in applications where the query distribution changes from time to time. When such a change occurs, the new algorithm would be able to revert back to a fast MTF strategy and then regain low query cost. It can then switch again to TR and try to fine tune the order of the list. By analogy with MTF⇒TR, we will refer to this new algorithm as MTF⇔TR.

Just like MTF⇒TR, the MTF⇔TR algorithm requires the gauging interval as a parameter. In fact, we can specify two different gauging intervals: one for deciding to switch from MTF to TR, and the other to revert from TR to MTF. In our experiments presented

here, we decided to use the exact same interval for switching in both directions. While still having the same problem of determining a good parameter value for the gauging interval, MTF⇔TR no longer suffers from the other drawback of MTF⇒TR, namely, that of the inability to cope with changes to the query distribution. We can now examine how the new algorithm stands up to MTF in a competitive setting.

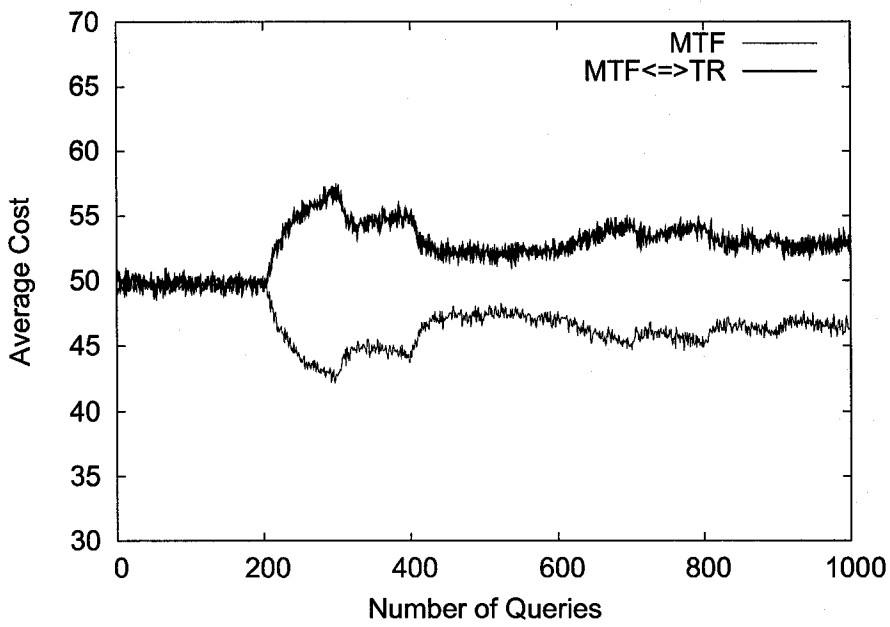

Figure 5.11: MTF versus MTF⇔TR in a competitive setting. MTF is still a clear winner.

Figure 5.11 shows the result of performing the same 10,000 simulation runs (100 runs for each of the 100 random query patterns) where each run consists of 1,000 simulated queries. It is clear from the figure that as soon as MTF⇔TR switches to a TR strategy it starts being outperformed by the plain MTF. After 100 queries (the length of the gauging interval) MTF⇔TR realizes that its average cost had increased and so it switches back to MTF. It manages to regain a little ground, but given that the opponent already had time to establish prevalence, the MTF scheme cannot catch up fast enough, and eventually decides to switch

back to TR whereupon it looses its ground again, and so on. In other words, any time that the MTF⇔TR scheme spends in the TR mode will be wasteful in the face of the MTF-only opponent. The clear conclusion here is that MTF is superior to a mixed MTF⇔TR strategy in a competitive setting.

5.5 Utilizing Knowledge of the Opponent

The Æip framework was designed to facilitate experimentation with entities that can be assembled in a variety of different experiments. In order to be able to use a given entity in as many experimental contexts as possible, we must strive to develop general-purpose entities that rely on as little domain information as possible[4]. This is precisely what we were doing with our Q-learning and DQ-learning based entities in Chapter 4, where the tic-tac-toe player entities and the LightsOut solver entity were effectively deprived of virtually all domain knowledge. This is also the approach that we have taken with our experiments with list-organizing strategies. As already emphasized in Section 5.4.1, the competing schemes have no knowledge of the actual list elements accessed (only the cost), let alone any knowledge of the opponent or even its existence. The assumption is that while a Scheme entity is trying to organize the list to its advantage, an external order-destroying force(s) may be present, whether simply interfering or fully adversarial. Under such an assumption, we can conclude that a plain MTF scheme is clearly superior to all others surveyed in this chapter.

In Chapter 4 we also considered the effect of providing some domain information to the learning entities. Not surprisingly, the player entities converge to an optimal strategy faster if knowledge of move alternation and legality is provided. We can also attempt such an experiment for competitive list organization. In a straightforward competition between TR

[4]That is not to say that we should not be able to provide as much domain information as we want to our entities. The only downside to such specialized entities is that they would be able to participate in a restricted number of experiments.

and MTF, the latter was shown to be a clear winner. In Section 5.4.3 we showed one way in which the ListWorld can level the playing field by balancing the mutually destructive reorganization operations. In this section, we present a different way of "boosting" the performance of the TR scheme, by allowing it to use the knowledge of the properties of the external order-destroying force, namely, the MTF opponent.

How can such knowledge be used? If the TR-based algorithm knows that the previous list reorganization was a move-to-rear (which is a move-to-front, from the point of view of the MTF opponent), it can negate the effect of such an operation by moving the rear element to its own front. We can call this new operation, RTF, or move rear-to-front. This new RTF operator is justified, as both opponents are facing the same underlying query distribution. To be specific, this new opponent of MTF would not only perform the TR operation but also an RTF operation. That is to say, the corresponding Scheme entity would emit an operator which is a combination of TR and RTF. We will refer to this new scheme as RTF+TR. For our new experiment, the Æip configuration as a whole and the alternation of queries, specifically, will remain the same as in all unbalanced competitions described earlier. It should be quite clear why we would expect RTF+TR to completely obliterate MTF in competition. Not only it would negate the previous organization operation of MTF but, also, turn the MTF's action to its advantage by transferring the element to its own "front" of the list.

Figure 5.12 shows the plot of the experimental results of a competition between MTF and RTF+TR. As usual, 10,000 simulation runs (100 runs for each of the 100 random query patterns) were executed where each run consisted of 1,000 simulated queries. As predicted, the combined RTF+TR scheme was a clear winner, reaching the low cost of ≈ 34, which is comparable to the performance of MTF against a plain TR as shown in Figure 5.6 earlier. In fact, given the destructive (to the opponent) nature of the RTF operation, it is not even necessary to perform a subsequent TR operation. The results would be exactly the same as in Figure 5.12. This also explains why RTF+TR scheme does not get closer to the optimal

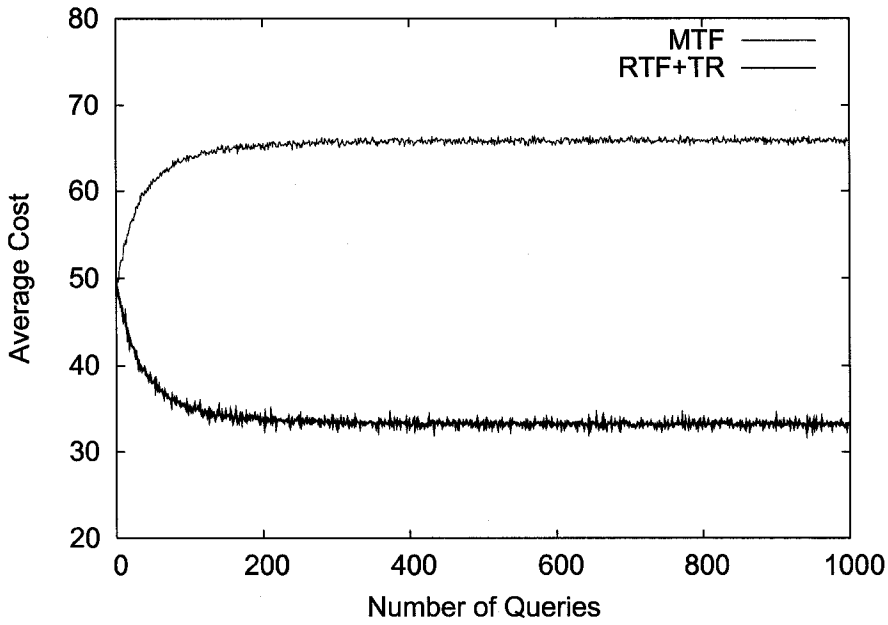

Figure 5.12: MTF versus RTF+TR in a competitive setting. In this case the knowledge of the opponent helps RTF+TR to overpower the opponent.

average access of ≈ 25 even though it includes the fine tuning that we can expect from the TR operation. If the list gets close to the optimal order, the single RTF operation is very likely to upset that order.

There may be several ways in which we can remedy the predicament of MTF when it competes against an RTF scheme. We can, for example, apply the same balancing approach we used in bringing about a "fairer" competition between MTF and TR. To apply balancing to this new experiment, we have to take into account the "destructiveness" of an RTF operation. It can also be measured as the number of equivalent transpositions that are required to move the rear element to the front of the list. Since the list size never changes throughout our experiment, this number of equivalent transpositions will always be the same as the size of the list less 1. Given that the TR operation contributes another 1 to

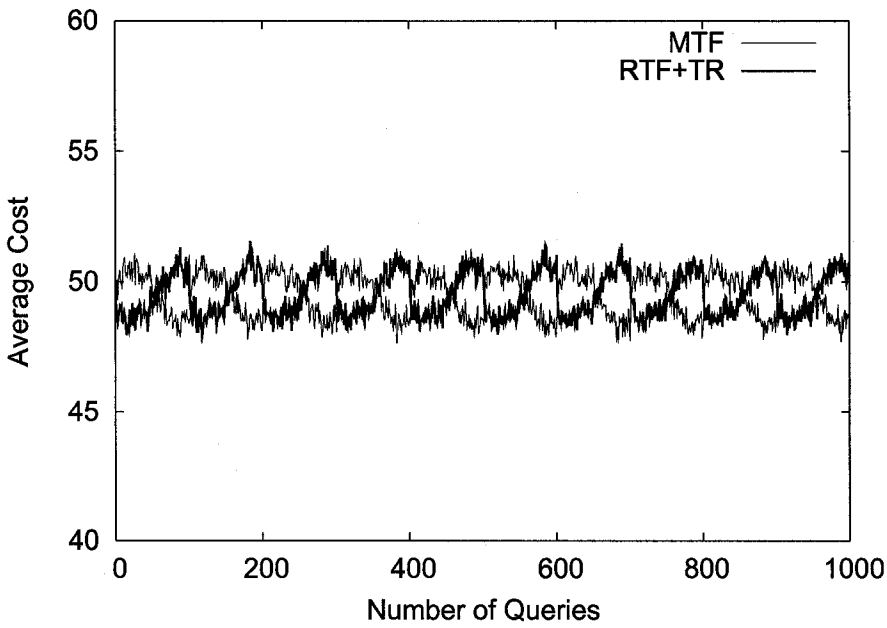

Figure 5.13: MTF versus RTF+TR in a balanced competitive setting. The two schemes overtake each other in succession.

the total, in the vast majority[5] of cases the number of equivalent transpositions for the RTF+TR combination will be equal to the size of the list. Figure 5.13 shows the results of such a balanced competition. We can see that neither scheme can offer an advantage as they overtake each other in succession in terms of the least average cost. Once again, we would like to emphasize that while balancing does equalize the chances of the schemes, we do not believe that in most applications it could be practical to use the balancing approach to counter the external order-destroying force. The experiments in this section serve the purpose of empirically confirming our earlier conjectures regarding the improved performance given the additional domain knowledge.

[5]Except when the queried element is already at the front, when no transposition is necessary.

Summary

One of the main aims of this chapter was to demonstrate how the Æip infrastructure allows us to easily set up novel experiments in a domain radically different from game playing and puzzle solving based on the Reinforcement Learning framework. The list organizing experiments reported in this chapter do not require either a reinforcement or feedback signal, yet can be just as easily conducted within Æip.

This chapter also offers a novel perspective on the problem of algorithm comparison. A variety of different criteria can be used to compare algorithms, yet these are usually difficult to combine in a single measure. We have offered a simple and attractive method for comparing adaptive algorithms by placing them in a direct competition with one another. We believe that this method can also be extended to comparison of non-adaptive algorithms (for example, of "sorting" algorithms) and that this is an interesting direction for future research. We realize that by providing the environment within which the algorithms compete, and, thus, by effectively providing the rules that determine the winner, we are creating a measure which *implicitly* combines multiple criteria for algorithm comparison. The main point is that such a measure may emerge naturally, and it may be easier to justify such a measure on an intuitive level.

Finally, this chapter contributes to the field of self-organizing data structures, by empirically demonstrating that a plain MTF scheme outperforms all other schemes that we experimented with in a competitive setting, where an external order-destroying force of unknown nature is present. If MTF can reach a lower average access cost in the presence of an adversary, it should do even better when that external force is less adversarial. For instance, if instead of ordering the list in opposite directions based on the same underlying query distribution, the two schemes were ordering the list in the same direction, but based on different distributions, we would expect MTF to do even better against such an opponent. The prevalence of MTF over all other schemes, including TR, in a competitive

setting is in sharp contrast with the widely accepted view that TR is a superior scheme in the traditional setting, assuming that the lower average cost (or amortized cost) rather than the speed of convergence is of overriding importance. We have also empirically confirmed the fact that the *reason* for MTF success is in the high degree of "order destruction" that each move-to-front operation subjects the opponent to. In other words, it appears that the MTF would be the fastest to recover from the element permutations caused by an unknown external source.

Chapter 6

Differentiated Robot Control using Æip

Introduction

In Section 3.2.5 we mentioned that the Æip framework allows us to set up novel experiments, where different aspects of the same system (for example, a robot) are controlled by several adaptive entities, either competing or cooperating with each other on a given task. If agents are cooperating with each other, the experiments would show how a complex system built out of simpler independent (or partially independent) subsystems can learn the novel environment. On the other hand, the competing agents would give rise to a situation where literally "one arm does not know what the other is doing". In this case, the experiments would be modeling a situation where the system must cope with some of its subcomponents either malfunctioning, or being disabled, or worse, taken over by an adversary. The purpose of this Chapter is to demonstrate in detail how such experiments might be setup, to propose algorithms that are capable of learning under these experimental conditions, and to discuss the obtained results.

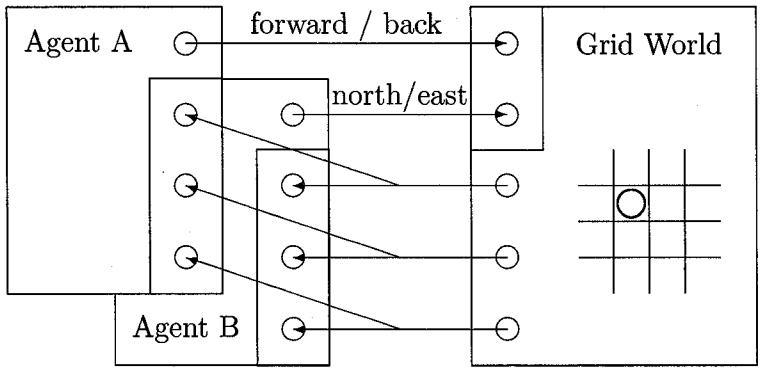

Figure 6.1: Two agents controlling different aspects of the same robot.

6.1 Possible Experimental Setups

Consider a simple grid world with a single robot moving from cell to cell. One could imagine an experiment where two or more agents are controlling different aspects of the same robot. Figure 6.1 is a variant of a setup similar to the one shown in Chapter 3, and it depicts the central idea behind the experiments described in this Chapter. Under the proposed setup, the Grid World entity is a rectangular two-dimensional space of known size with a robot rolling inside it on wheels. The premise of the setup is that the robot can turn its wheels only in fixed directions, e.g. north and west. Given a certain direction of the wheels the robot can choose to go either forward or back. Figure 6.1 shows that these two independent decisions (wheel direction and motion) are controlled by two different agents.

6.2 Experiments

Given that we already had an implemented Grid World entity from other experiments, we wished to reuse this implementation. In doing so, we encountered a minor incompatibility problem, where the existing Grid World entity featured a single inport (robot action), which expected a direction (north, south, east, or west) signifying where the robot is to move next.

In contrast, in the context that we required, the world accepts two independent actions, i.e. the wheel direction and forward/backward motion. The Æip framework is flexible enough, however, to address this incompatibility. We have included this point here specifically to highlight the ease with which we can solve the problem without going outside the boundaries of Æip.

The solution in this case is to introduce an intermediary entity, which we called the Controller, that combines the inputs from the two agents and transforms them into a single robot action fed into the corresponding inport of the Grid World. The matching Æip diagram is shown in Figure 6.2. The same Figure also shows additional implementation details, namely the presence of two other signals: *move count* and *end of episode*. The latter indicates when a training episode ends, which usually means that some goal in the environment had been reached. In the context of the Grid World that we used, it means that a particular goal cell had been reached. The move count signal is meant to provide a sense of time to the agents solving n-ary tasks, such as reaching the goal cell within a fixed time limit.

6.2.1 Q-Learning with Independent Agents

Our first attempt to solve this Differentiated Control Problem was with a pair of regular Q-Learning agents. The Q-Learning algorithm is guaranteed to converge only when facing an environment possessing the Markov property. In this case, however, the environment for each of the agents clearly violates the Markov property when the agents have not converged yet. We, therefore, started these experiments with an expectation of non-convergence. It turned out, however, that under certain conditions, the agents do converge, albeit to suboptimal solutions. If the Markov condition was satisfied, we would expect convergence to optimal solutions (i.e. shortest paths) for every starting state.

As the task of finding a goal cell is of limited horizon in nature (each episode is terminated when the goal is found), we decided not to discount any of the reinforcements, i.e. the

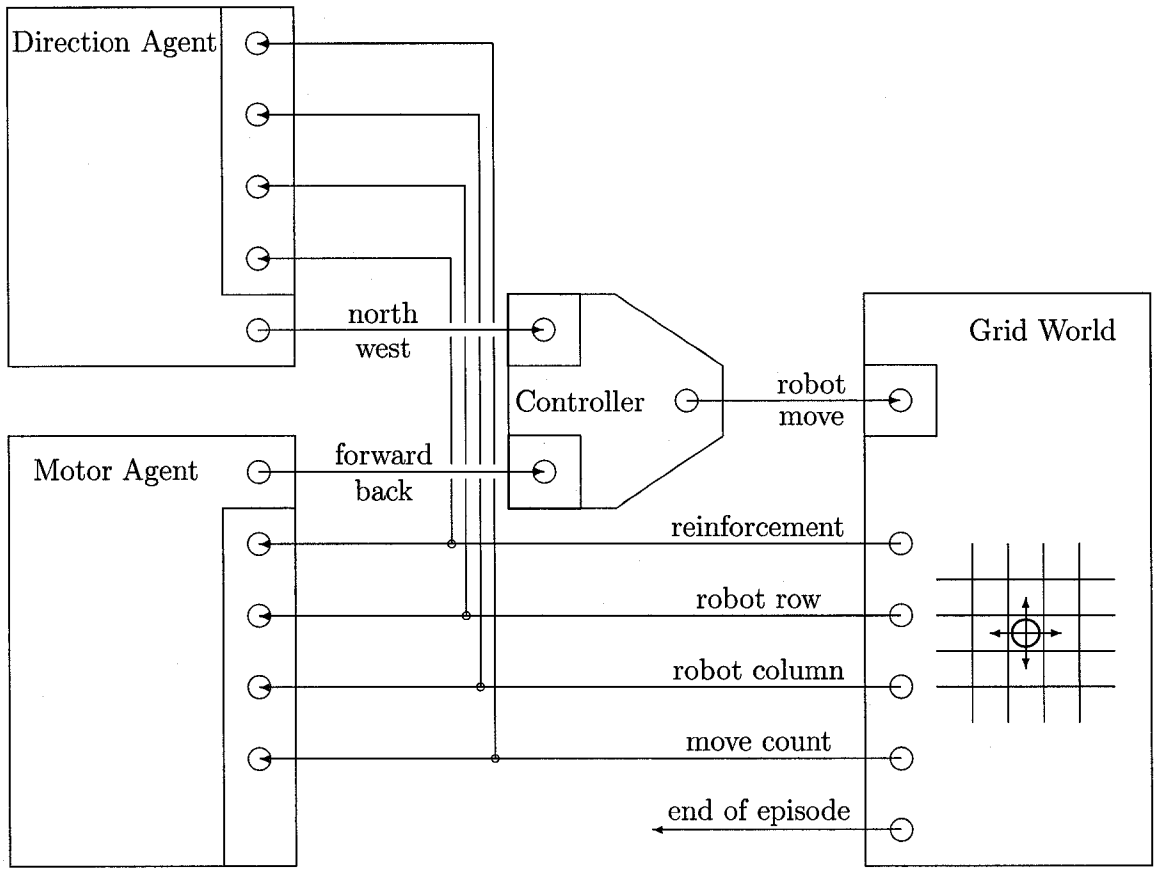

Figure 6.2: The Controller entity combines the actions of two independent agents into a single signal that is understood by the Grid World.

discount factor was set to 1. For initial experiments the learning rate was also set to 1 to encourage faster convergence of both agents. We chose a standard reinforcement function which penalizes every move with a -1 reinforcement. This biases the agent towards shorter episodes (least amount of punishment), and hence to seek shorter paths to the goal. If all the Q values are initially set to 0, such a reinforcement function in tandem with a greedy action-selection strategy (temperature set to 0) naturally encourages exploration, where unexplored paths would have higher initial returns (0 as opposed to some negative number).

We ran the experiments for grids of size 10×10, 100×100, and 1000×1000 and, quite surprisingly, observed convergence in all of these cases, albeit convergence to somewhat suboptimal solutions. Each of the following figures depicts the results of averaging several independent runs, each of which were consisting of a sequence of learning episodes. Each such sequence was produced with different sets of random seeds (one seed for the Grid World, and another seed for each of the two learning agents). In all experiments, the goal cell was situated in one corner of the grid, whereas the robot was randomly positioned at the outset of every learning trial.

Figure 6.3 plots the average solution length (averaged over 1000 independent runs) against the number of learning trials for a 10×10 grid. Note the logarithmic scale of the vertical axis. Given that the robot is equally likely to start the episode in any of the 100 cells, the optimal expected length of path to goal should be exactly 9 moves, as if the robot was always starting from the middle of the grid. Our Grid World entity, however, is coded in such a way as to never position the robot right into the goal cell at the outset of the experiment. This affects the optimal expected length only marginally as it now becomes $9 \times 100/(100 - 1) \approx 9.1$. Although it may not be clear from the graph, the agents converged to an average path length of approximately 9.6, which is close to but not optimal. At first it may seem like this difference is not statistically significant, but as we shall see in the following experiments it actually is, i.e. the agents do fail to find the optimal policy. On the average, it takes only about 200 episodes to converge.

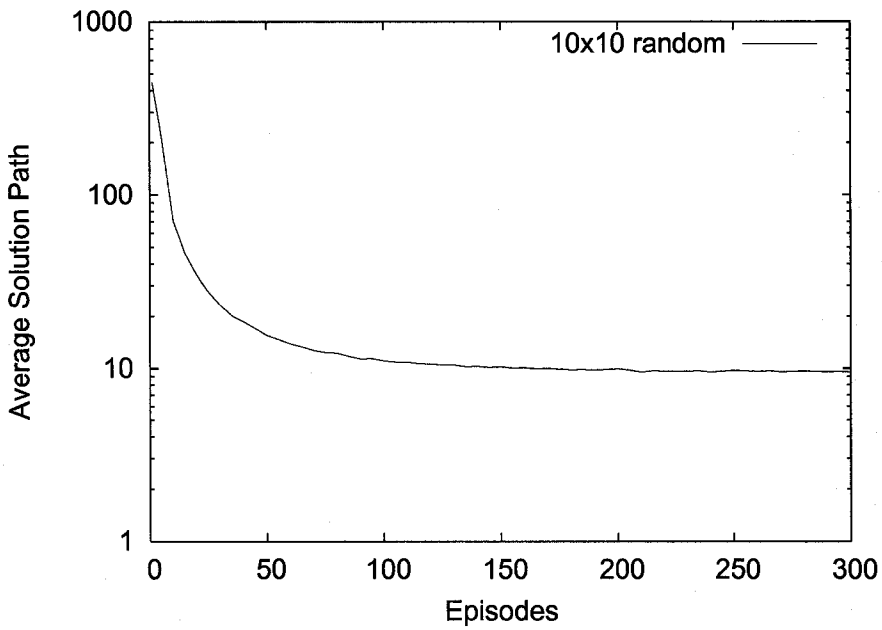

Figure 6.3: Q-learning agents converge to nearly optimal solutions for 10×10 grid size.

Figure 6.4 plots the average solution length (averaged over 100 independent runs) against the number of learning trials for a 100×100 grid. As before, the optimal expected path length should be $99 \times 10000/(10000 - 1) \approx 99.01$, while the experiments show an average of approximately 111, once again falling short of the optimal. On the average, it takes approximately 10,000 episodes to converge to this suboptimal path length.

Figure 6.5 plots the average solution length against the number of learning trials for a 1000×1000 grid. Due to the length of time it takes for one sequence of episodes to complete (initially it takes a random-walking robot over 5 million moves to stumble across the goal cell), it took several days of CPU time to average over 10 independent runs. The optimal expected path length for this grid size is practically the same as 999, yet the figure shows premature convergence to approximately 1200 after an average of a half a million of episodes.

The key ingredient to the surprising convergence in all of the above cases seems to be

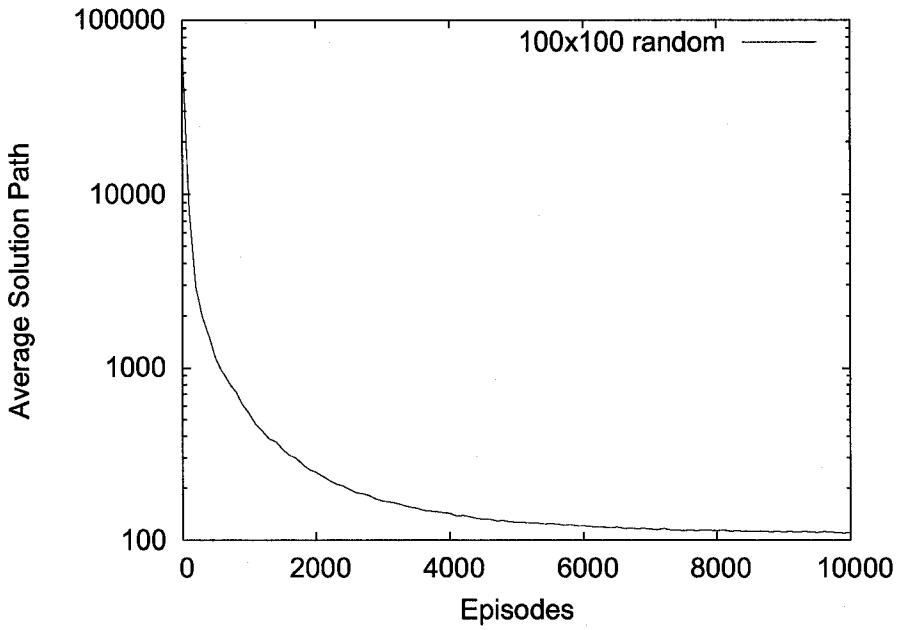

Figure 6.4: Q-learning agents converge to suboptimal solutions for 100×100 grid size.

the fact that each learning trial begins by positioning the robot randomly in the grid. If, instead, every trial begins with the robot positioned in some fixed location, the combined two-agent system does not necessarily show the same convergence. Much seems to depend on the chosen fixed starting position of the robot relative to the goal cell, and a particular choice of the random seed.

For comparison purposes, Figure 6.6 plots the average solution length for a 10×10 grid, averaged over 1000 independent runs. Each trial started with the robot at the coordinates (0,9) while the goal cell was in the adjacent corner with the coordinates (9,9) like in the other experiments. Again we can observe a convergence to a suboptimal value of approximately 11.6 while the optimal length is 9.

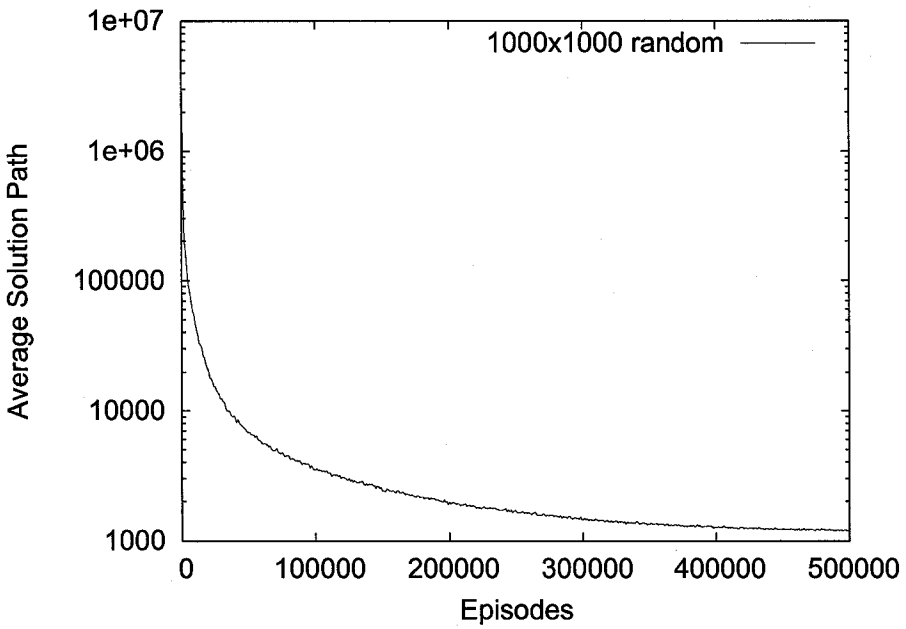

Figure 6.5: Q-learning agents converge to suboptimal solutions for 1000×1000 grid size.

6.2.2 Q-Learning with Communicating Agents

The problem with the above experiments is the fact that the two controlling agents act completely independently of each other, and more importantly are completely unaware of each other's action choices. Can we somehow preserve the differentiated control nature of the experiments and yet observe optimal convergence to shortest paths? One idea is to allow one of the agents to observe the action chosen by the other *before* committing to it's own decision. This is the most primitive way by which the two agents can be said to "communicate" with each other.

Figure 6.7 is the corresponding modification of the original Æip diagram in Figure 6.2. The figure shows an additional inport in the Motor Agent, accepting the direction signal from the Direction Agent. In this configuration, the latter makes the decision about the direction

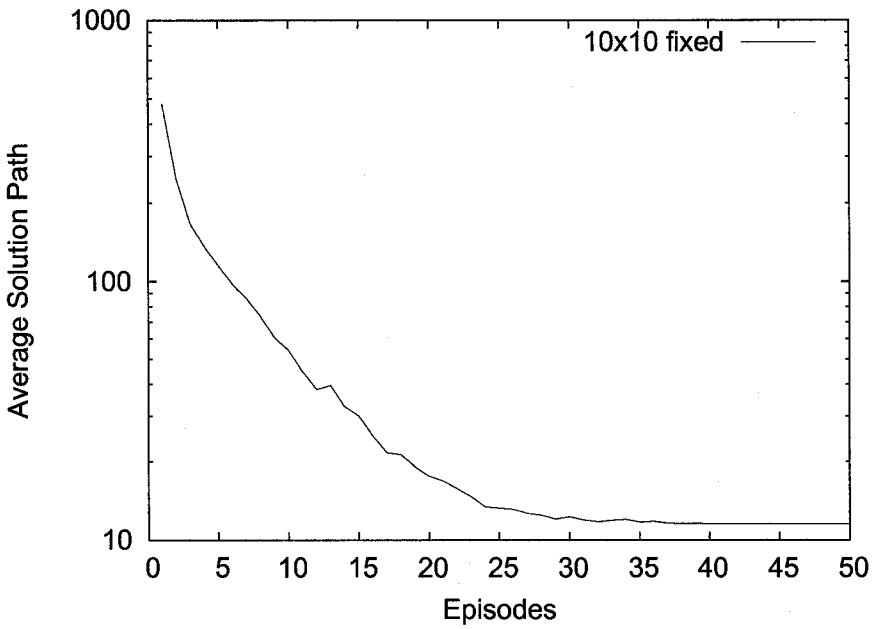

Figure 6.6: Q-learning agents converge to a suboptimal policy if the robot always starts in the same position on the grid. Grid size: 10×10.

(north or west) first, and then the Motor agent is allowed to select the forward or backward motion in that direction. As usual, however, neither of the agents have any "understanding" of *what the inport values represent*, including the chosen action of the Direction Agent. Both agents learn the consequences of their decisions by observing the ultimate changes in other inports, which represent the position of the controlled robot.

Such a configuration can be viewed as an acquired rudimentary one-way communication between the agents who, through interacting with the environment, learn the benefits of such communication in solving a given task of finding the shortest path to a goal cell. Since we want one agent to choose an action based on the decision of the other agent, we need to change the interaction schedule of the entities. In the experiments described below, one of the agents acts on the first clock tick, followed by the other agent on the second tick and,

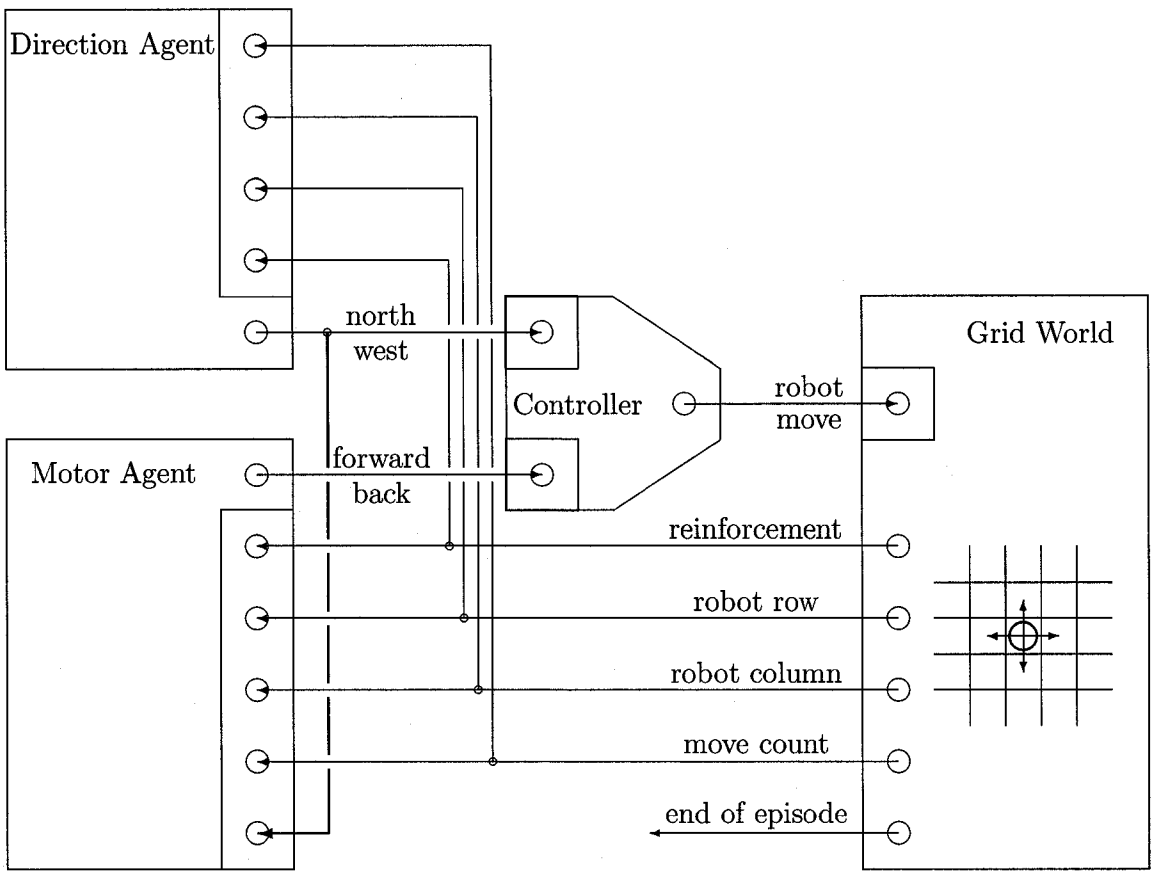

Figure 6.7: Here the Motor Agent observes the chosen direction of the Direction Agent before committing to its own decision whether to go forward or backward.

finally, followed by the Grid World entity's response on the third tick. Just as before, the agents and the Grid World alternate, except this time the two agents no longer choose their actions simultaneously.

As before, both the discount factor and the learning rate were set to 1. The reinforcement function (which penalizes every move with a -1 reinforcement) was also left the same in this setting of "communicating" agents. In contrast to the previous experiments, though, we did not position the robot randomly at the beginning of each learning trial. Instead, the robot was always positioned in the corner opposite to the goal. For example, in a 10×10 grid, the robot always started at coordinates (0,0) while the goal was always at (9,9).

We ran the experiments for the same grid sizes (10×10, 100×100 and 1000×1000), and observed convergence to optimal length solutions in all of these cases. For the 10×10 grid, each experiment ends fairly quickly and so the plot presents a curve averaged over 1000 runs, each run initiated with a different random number generator seed. Due to experiments taking longer time to converge, for a 100×100 grid we averaged over a 100 runs. However, for the 1000×1000 grid, a single experiment takes about a day of CPU time - the very first trial, for example, being essentially a random walk, lasts for over 7 million robot moves. Therefore, this scenario we averaged over only 10 runs.

Figure 6.8 plots on the logarithmic scale the average solution length (averaged over 1000 independent runs) against the number of learning trials for a 10×10 grid. Note that for a 10×10 grid the optimal length path to goal is $(10 - 1) + (10 - 1) = 18$ robot moves — which is exactly what the plot shows — converging to this length after only a *100 episodes* on average.

Figure 6.9 plots the average solution length (averaged over 100 independent runs) against the number of learning trials for a 100×100 grid. The optimal length for this grid is $2 \times (100 - 1) = 198$. The experiment results in convergence to this number as the plot in the figure demonstrates, after approximately 7500 episodes, on average.

Figure 6.10 plots the average solution length against the number of learning trials for a

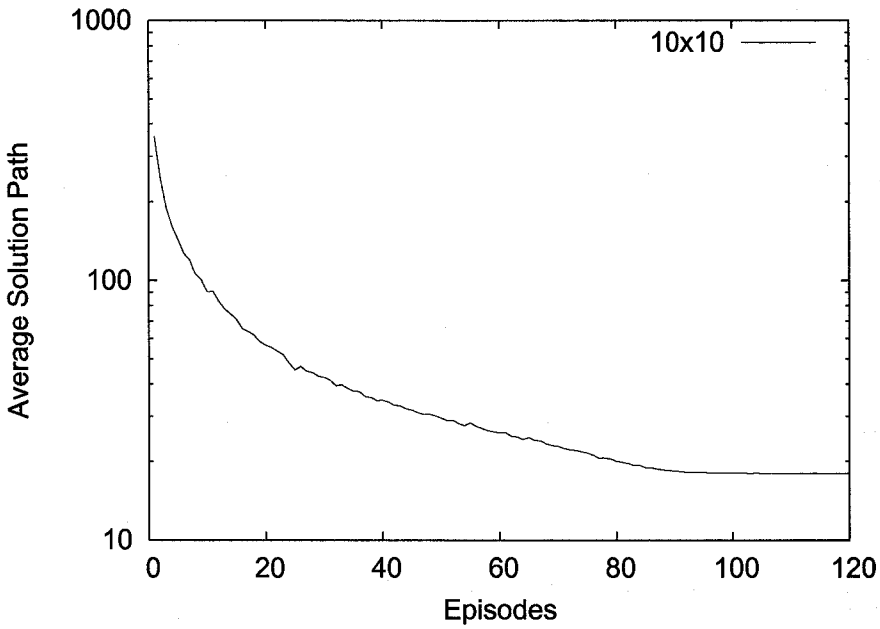

Figure 6.8: Q-learning agents converge to optimal solutions for 10×10 grid size if one agent observes the decision of the other agent.

1000×1000 grid. As mentioned earlier, this plot shows convergence averaged over only 10 runs, resulting in the curve being not as smooth as the other two. It clearly demonstrates convergence to the optimal value of $2 \times (1000 - 1) = 1998$.

It is worth noting that the plots in Figures 6.8, 6.9, and 6.10 exhibit a curious convergence pattern not seen in ordinary experiments of Section 6.2.1, namely that of a sudden acceleration in convergence close to the optimal path length, effectively creating an inflection point in the curves. While not well pronounced in the 10×10 plot, it is quite clearly visible in the two other plots, especially the 100×100 one. In particular, Figure 6.9 shows the inflection point around 6,500 episodes, about 1,000 episodes before final convergence to the optimal path length. We believe that this phenomenon is, at least, in part caused by the fact that both agents are simultaneously improving, and when both are close to converging

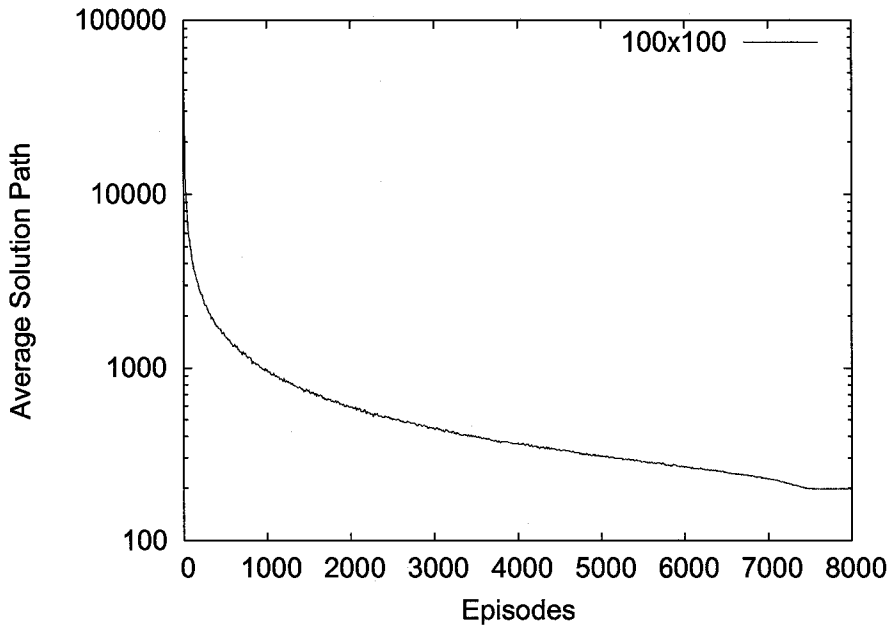

Figure 6.9: Q-learning agents converge to optimal solutions for 100 × 100 grid if one agent observes the decision of the other agent.

to mutually optimal policies, they are much more likely to jointly choose an optimal action. The reader should keep in mind that in all of the experiments in this Chapter, the action-selection policy was greedy (i.e. , the temperature was set to 0), and thus the agents always chose an action that appears to be best given what is known at the time.

6.2.2.1 "Direction" and "Motor" Agents Switched

Interestingly, if we switch the agents around (that is, if the agent that controls the direction of the robot now observes the action of the motor-controlling agent before committing to its own), the same phenomenon of accelerated convergence is no longer present. Moreover, it takes the tandem of agents much longer to converge. In the case of 10 × 10 grid, albeit very close to the optimal 18, it turns out that the agents don't quite converge to 18 even

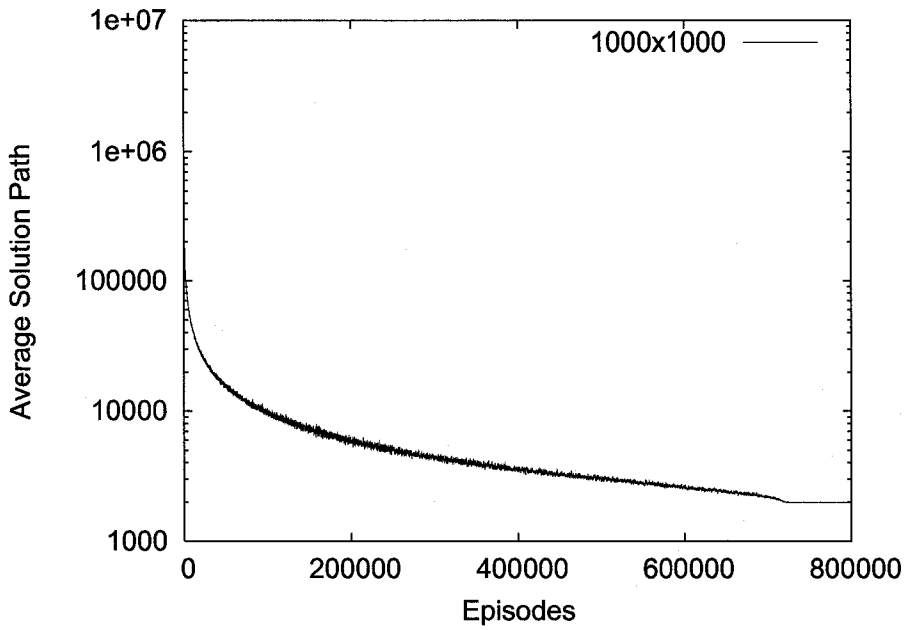

Figure 6.10: Q-learning agents converge to suboptimal solutions for 1000×1000 grid size.

after 500 episodes (which must be seen in contrast to the 100 episodes previously). Slower convergence is equally apparent in the case of 100×100 grid, the convergence plot for which is shown in Figure 6.11. Although it may not be very clear due to the logarithmic scale of the plot, it barely converges after 20,000 episodes (also averaged over a 100 independent runs), and thus very slowly approaches the optimal of 198. Instead of accelerated convergence, we have a very long and drawn out convergence without any obvious inflection points.

We should point out that our setup does contain an inherent asymmetry with respect to the capabilities of the two agents to control the robot. The experiments clearly show that knowledge of direction before committing to motor action, is much more valuable during learning than the other way around. This was an unexpected outcome as it is not at all obvious why this should be the case.

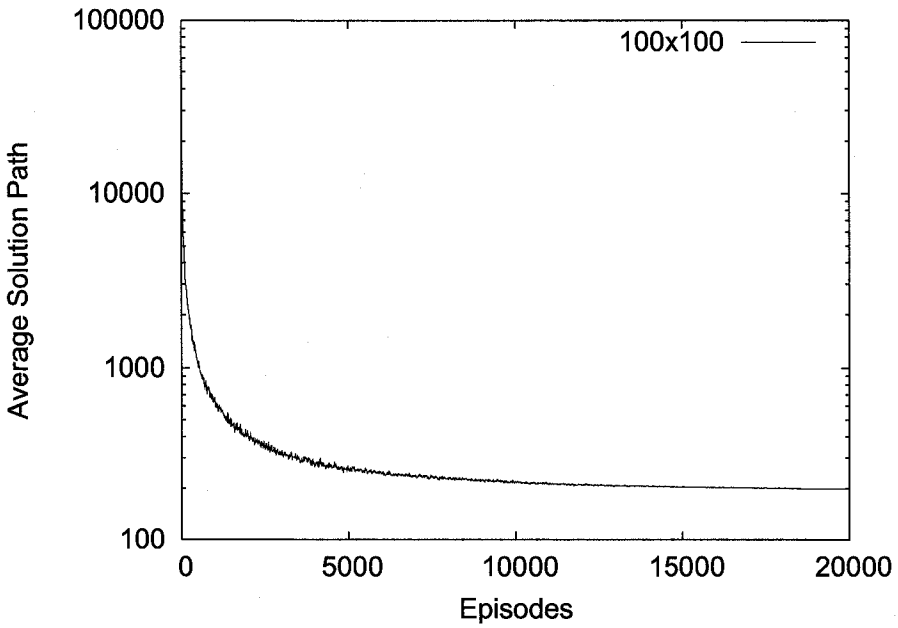

Figure 6.11: Q-learning agents converge slower if the "direction" agent observes the "motor" agent.

6.2.2.2 Communicating Agents Make Simultaneous Decisions

Another variation of the experiment we attempted was a modification of the interaction of the involved entities, so that both agents select their actions simultaneously. This has the effect of the "motor" agent observing the **previous** decision of the "direction" agent, rather than the most relevant, i.e. the current action. It would seem that such an observation would be utterly useless to the "motor" agent, but as our experiments indicate, this is not the case. As it turns out, the previous decision is a reasonable predictor even in a situation when both agents learn (and thus change their policies) simultaneously.

For instance, Figure 6.12 appears to be virtually identical to the corresponding curve in Figure 6.9 (it is also averaged over 100 independent runs). The same exact accelerated

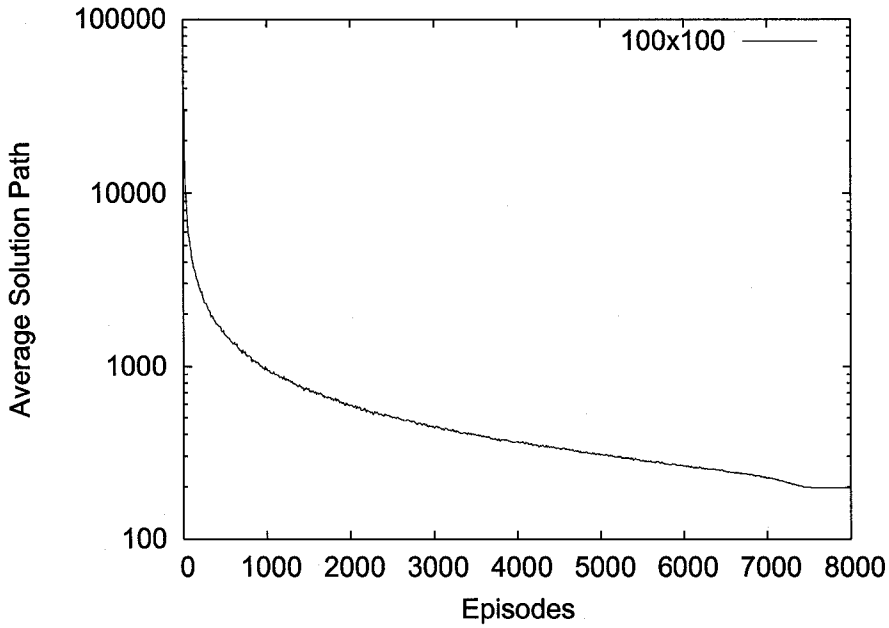

Figure 6.12: Q-learning agents appear to be converging when both agents make simultaneous decisions.

convergence leads to an apparent plateau at 198. Upon closer examination of the numbers, however, we realize that a small percentage of runs converge to a suboptimal path length and consequently the averaged curve does not quite reach the optimal — even if it approaches the optimal quite closely. It is, perhaps, not surprising that such a setup manages to approach the optimal, considering that the non-communicating agents were capable of that as well. What is surprising, is just how close the agents approach the optimum, given our *a priori* expectations of this setup.

6.2.2.3 Communicating Agents Using Discretized Q-learning

As a final variation on the theme, we would like to demonstrate how Discretized Q-learning can be used to modify these experiments somewhat. For instance, we can insist that episodes

do not progress beyond a certain fixed number of steps. In other words, we can put an upper bound on the length of paths to goal. If the robot does not reach the goal in less than this fixed number of moves, the episode is ended, the robot is repositioned back to the starting cell, and a new episode begins.

In order to accomplish this, we can keep the same exact setup where the "motor" agent observes the decisions of the "direction" agent. Since the agents are now using the Discretized Q-learning algorithms, we need to supply them with the goal arity. In the experiments reported here, the number of moves was capped at 50. If we prefer shorter paths over longer paths, we are clearly dealing with a goal arity of 51. The only other modification required was to ensure that the reinforcement signal that corresponds to the requisite feedback signal is provided to the agents in an integer form, i.e. discrete.

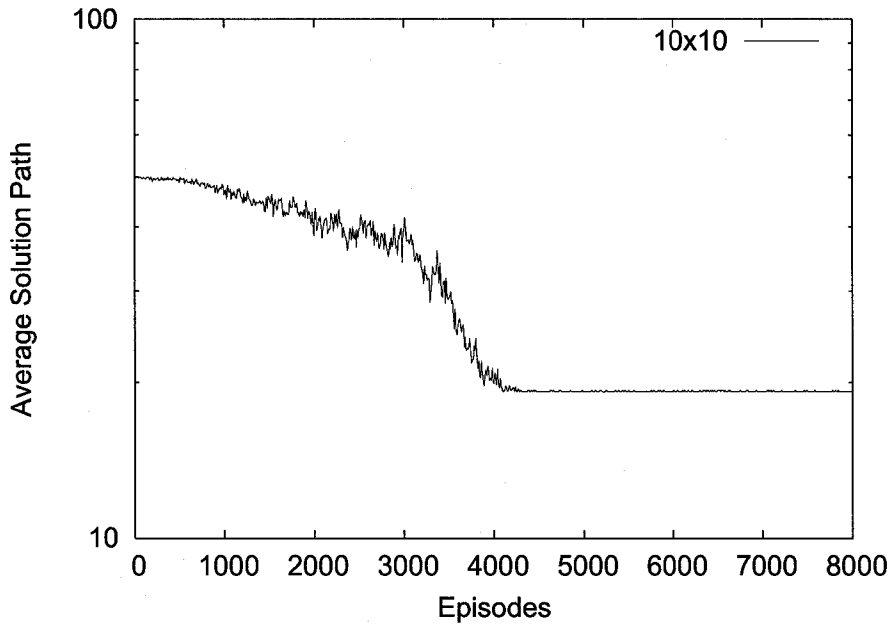

Figure 6.13: Discretized Q-learning agents approach near optimal solutions on average.

Figure 6.13 shows the convergence plot for a grid of 10×10 averaged over 10 independent

runs. The average stabilizes around 19 moves whereas the optimal path length is 18. The individual runs converge to different values, such as 18, 19, or 21, etc. We thus infer that in the case of limited episode length, even the fact that the agents communicate does not guarantee convergence to the optimum.

Summary

In this Chapter we showed how the Æip framework can be used to setup interesting novel experiments, whether using "off-the-shelf" Q-learning or the Discretized variant. The experiments show that convergence to near optimal values is not difficult to achieve in the chosen Grid World setup even if the robot is independently controlled by two distinct and simultaneously learning agents. Convergence to optimal values is possible if one of the controlling agents can see the decisions made by the other before committing to its own. The experiments also show that the controlling capabilities of individual agents play an important role in the speed of convergence.

Chapter 7

Conclusion

7.1 Æip and JAGUAR

In this thesis we have described an Agent-Environment interaction protocol, or Æip, as a generalization of the agent-environment feedback loop, based on the notion of interconnected entities. While, in itself, this is not the most important contribution of this thesis, it provides the foundational cornerstone for all other contributions. We refer to Æip as a *protocol* because we advocate the importance of standardization of communication between the environments (whether simulated or real) and the learning agents. The protocol can trace its roots to [11].

Our hope is that the JAGUAR experimentation tool (as a reference implementation of Æip) will prove successful, and that it will be accepted by the research community. At issue is the ability of AI researchers to quickly set up an experiment using components drawn out of *standard* libraries of configurable *entities*. The easier it is to set up the experiments, the more experiments one can try out and, therefore, the more productive a researcher becomes. Indeed, if various libraries and toolkits are in existence, and they are not inter-operable (as different researchers may have their own implementations), the corresponding systems become completely incompatible with one another. If Æip is implemented as a network

protocol, the individual entities themselves do not need to be running on the same platform, computer, or even implemented in the same language.

The most important contributions of this thesis arose from our attempts to see commonality in, and to question assumptions behind, existing frameworks within which Interactive Learning is studied.

7.1.1 Future Work

Now that JAGUAR has been turned into an open-source project, we would like to continue work on the tool. In order to aide in the wide adoption of the tool, we can make the interface more intuitive and user-friendly. We also need to expand the library of available agents and environments, and we are hoping that the open-source nature of the tool will naturally promote such expansion, leveraging the creativity of the community of users. Additionally, we can attempt to take JAGUAR in other promising directions, such as the ability to conduct distributed experiments where the interacting entities reside and execute on physically different computers separated by a network.

We are also convinced that continued work on and with the tool will expose the limitations not only in the tool itself, but also in the Agent-Environment interaction protocol that the tool embodies. For instance, we would like to consider extending the protocol to the continuous (non-discrete) simulations where we would need to deal with the challenge of all entities changing their states asynchronously. Just as the contributions of this thesis were, to a large extent, borne out of our frustrations with setting up experiments, we hope that further experiments will likewise lead to new ideas.

7.2 Playing Games Without Knowing the Rules

The first contribution resulting from Æip is the series of experiments on learning to play a game under the extreme conditions of minimal *a priori* information. While the experiments where the agent was asked to learn some rules of the game are not new, to the best of our knowledge, agents were never before given as little information as in our study. Not only did the learning entity not know what the best move for a given position is, it also did not "know" what constituted a legal move, whether it was playing against an opponent, and, hence, whether opponents must alternate moves. Some preliminary results of this study are published in [13].

7.3 Goal Analysis and Feedback Signal

The next important contribution was our addressing the question of whether the reinforcement signal, so popular in Learning by Interaction, is the best, or the only way of providing the goal information to the agent. To this end, we had to ponder the meaning of goals (or tasks) and, consequently, developed a novel methodology of classifying goals according to their *arity*. As a result, we have proposed the *feedback* signal as an alternative way of supplying goals to the agents relative to our arity-based classification of tasks. We have demonstrated the viability of the feedback signal by providing a discretized version of Q-learning with an unexpected property of being able to utilize a *single* bit of information per table entry to solve a binary non-deterministic MDPs. These results are explained in Chapter 3 and published in [12]. Goal classification, additionally, helped us to identify a "difficult" category of tasks that we termed *exploratory*.

7.3.1 Future Work

We would like to improve our understanding of goals by considering:

- probabilistic goal definitions, e.g. the agent succeeds if it chooses an acceptable trajectory 90% of the time,
- continuous trajectories through universal state space, and
- ways of specifying exploratory goals.

All of these directions are bound to yield interesting new findings, but are also likely to require much more work.

7.4 FQ-learning and its Convergence

From a theoretical point of view, the most important contribution is the development and proof of convergence (in the same Chapter 3) of FQ-learning — a feedback based derivative of Q-learning, which relies on a greedy action-selection policy. We succeeded in showing that for a class of learning tasks (specifically, deterministic, Markov, episodic, binary tasks), it is sufficient to use FQ-learning in order to have a guaranteed finite-step convergence to a solution. This result is stronger than the corresponding result for Q-learning for two reasons. First, FQ-learning no longer requires that the algorithm observe each state-action pair infinitely often. This is a rather inconvenient requirement to deal with in practice: either it cannot be satisfied or the algorithm is forced to continue making suboptimal decisions even after convergence. FQ-learning is based on an absolutely greedy action-selection policy. This has the added advantage of the algorithm "behaving", throughout the learning phase, as optimally as its current knowledge permits. Second, the classic Q-learning is only guaranteed to converge asymptotically, whereas the duration of the learning phase of FQ-learning is bound by the total number of states it can observe. It is also possible (and, in fact, desirable) that FQ-learning will converge before seeing every state-action pair. The latter is possible due to the overestimate nature of the feedback signal.

7.4.1 Future Work

Development of additional theoretical results appears to be promising and is likely to be an objective of a shorter range. We believe that the result embodied in our FQ-learning convergence theorem can be further extended to general N-ary tasks (still Markovian), to non-deterministic Markovian tasks (given the non-probabilistic definition of acceptability), and potentially to even non-episodic tasks. Our future work will attempt to address these cases.

While FQ- and DQ-learning are Q-learning derivatives intended to work with the feedback signal, we may be able to develop algorithms that were designed to work with feedback from the very beginning, taking advantage of the known arity of the task. For instance, if we know the task to be binary, could we construct an algorithm that simply records the unacceptable trajectories it has seen so far and then eventually store only the one acceptable trajectory that it discovered? Or, could we also develop a successful binary-goal algorithm which converges even when the environment does not satisfy the Markov property? These are very interesting questions that are worth investigating.

7.5 Applying DQ-learning to "Large" Problems

The solution to several variants of the LightsOut puzzle (Chapter 4) also clearly demonstrates that the discretized form of Q-learning can tackle problems with large state spaces, truly making a difference between being able to and not being able to solve a problem through Interactive Learning. We have demonstrated that DQ-learning is equally successful at solving puzzle variants regardless of whether analytical solutions to these puzzles exist. We have also used the puzzle domain to examine and compare DQ-learning performance when dealing with goals of different arity. The successful application of DQ-learning to LightsOut is published in [14].

7.5.1 Future Work

We believe that discretization has not been considered within the context of RL, and that much can be done to investigate discretized versions of algorithms other than Q-learning. Just as the discretization of existing LA algorithms resulted in much innovation, we expect similar benefits to be reaped from systematically considering discretized variants of RL algorithms. We see algorithms where the Markov property is violated, such as POMDPs (Partially Observable Markov Decision Processes), as prime candidates for such discretization. General Temporal-Difference methods (of which Q-learning is a special case) are likewise attractive targets. Moreover, alternative ways of performing discretization should be investigated. In particular, what if in the ordinary Q-learning, the Q value was allowed to jump in discrete increments only, while the reinforcement remained to be a floating-point value of arbitrary precision?

Another avenue in which our work can be easily extended is the application of DQ-learning and the corresponding memory savings to new problem domains. Problems that were not previously solvable due to memory requirements can now be attempted by using the thrifty memory representation of DQ-learning. As computers with more and more memory become available to researchers, the "horizon" of solvable problems expands. DQ-learning gives an extra bit of advantage in such an expansion. We have already started to experiment with applying DQ-learning to Tetris, and even contemplated finding the full solution to checkers.

7.6 Novel Experiments

In Chapters 5 and 6 we have shown how the Æip infrastructure allows us to easily set up novel experiments in a domain radically different from game playing and puzzle solving. The list organizing experiments reported in Chapter 5 do not require either a reinforcement or

feedback signal, yet can be just as easily conducted within Æip.

In Chapter 5 we also offer a simple and attractive method for comparing adaptive algorithms by placing them in direct competition with one another. We realize that by providing the environment within which the algorithms compete, and, thus, by effectively providing the rules that determine the winner, we are creating a measure which *implicitly* combines multiple criteria for the comparison of algorithms. The same chapter contributes to the field of self-organizing data structures, by empirically demonstrating that a straightforward MTF scheme outperforms all other schemes that we experimented with in a competitive setting, where an external order-destroying force of unknown nature is present. The prevalence of MTF over all other schemes (including TR) in a competitive setting, is in sharp contrast with the widely accepted view that TR is a superior scheme in the traditional setting. In other words, it appears as if the MTF would be the fastest to recover from the element permutations caused by an unknown external source.

In Chapter 6 we showed how the Æip framework can be used to setup interesting novel experiments where agents show rudimentary cooperation and communication capabilities. The experiments show that it is possible to achieve near-optimal convergence, even if the robot is independently controlled by two distinct and simultaneously learning agents. Convergence to optimal values is possible if one of the controlling agents can see the decisions made by the other before committing to its own. The experiments also show that the controlling capabilities of individual agents play an important role in the speed of convergence.

7.6.1 Future Work

In Chapter 5 we introduced a method for comparing adaptive list-organizing algorithms in a competitive setting. We believe that this method can also be extended to comparison of non-adaptive algorithms such as the "sorting" algorithms. This is an interesting tangential direction for future research.

Finally, it would be interesting to extend the work described in Chapter 6 by explore multi-agent experiments where inter-agent two-way communication can emerge as a behaviour learned in the self-interest of each agent. This is just one example of the many novel experiments that can be attempted.

In conclusion, all of the above contributions were borne out of our generalization of the agent-environment interaction. We believe that this thesis demonstrates this generalization to be fruitful. In particular, we have demonstrated its theoretical merit and its applicability in an ensemble of areas including game playing, organization of data in adaptive lists, and differentiated robot control. We expect that other important contributions can result from this work.

Appendix A

AI Philosophy

A.1 The Plight of AI

The field of Artificial Intelligence has a reasonably long history. In spite of this, even today, a thorough, all-encompassing formal theory of intelligence is lacking. Surprisingly, very few attempts are being made to formalize this notion. Most of the current research proceeds from a simple premise: If we provide a Computer Science solution to a problem that usually requires *human* intelligence, we must be building an artificially intelligent system. This is a pragmatic approach that does not require a definition of intelligence as a property of a system, just the ability to recognize what constitutes such problems[1].

The difficulty in bringing objective and precise metrics to the study of intelligent behaviour has long been recognized and best exemplified by a quote from an article entitled "In memory of Ivan Petrovich Pavlov" written by the famous Soviet physicist and Nobel Prize winner, Pyotr Kapitsa[2]:

[1]Although many of the statements we make here are arguable, they should not detract from the thesis and the problems we endeavour to study.

[2]The article appeared in the Soviet newspaper "Pravda" in 1936. Translated from Russian by the author.

> As the founder of a whole new area in physiology, Pavlov has achieved a remarkable renown. It is difficult, for us physicists, to comprehend the full depth and finesse of his work, but there exists one aspect that unites our fields. In physics, we cultivate quantitative and precise measurement methods, considering them as one of the most important means of uncovering the essence of studied phenomena. Many areas of physiology have not yet adopted methods of precise measurement, and it would appear that the study of advanced nervous activity represents the most complex and arduous field for their adoption. Nevertheless, Pavlov found objective and quantitative methods of measuring and assessing the psychological phenomena, this being one of his enormous scientific triumphs.

Pavlov did not explicitly address intelligence, but his initial success was followed by many psychological theories of intelligence (see [80] for a survey), which, though powerful, could be viewed as possessing a number of shortcomings from an AI perspective. First of all, they typically focus on one specific "hardware" — that is, they study the human intelligence. Secondly, such theories are rarely mathematically inclined, which is a prerequisite for any formalization.

It is our position, that until such a formal theory is proposed, the science of Artificial Intelligence will not reach maturity. Without a well-founded understanding of precisely which properties are required for intelligent behaviour, and to what quantifiable extent these properties must be present, we will never be able to conclude with a sufficient degree of certainty, that we have constructed an "intelligent" system.

While this thesis does not offer such a theory of intelligence, it provides a modest step in this direction — a framework within which such theories may be developed, prototyped and, hopefully, tested.

A.2 AI Philosophy

After the initial successes in AI a few decades ago, many leading scientists in the field became proponents of a viewpoint, that came to be known as "strong" AI[3]. In contrast to the "other" or "weak" AI[4], the strong AI supporters claim that a running computer program, that performs symbolic manipulations and appears to behave in a human-like fashion, must, in fact, possess the human qualities of understanding, self-awareness and intentionality. In other words, it must really "think". This claim is even stronger than the one envisioned by the famous Turing test, as Turing himself was quite careful in avoiding the question of whether machines can "think". Instead, he addressed a much less controversial question of whether machines can ever be capable of "imitating" a man.

The claim of strong AI has been attacked by many. Armed with a variety of arguments, Searle [75] and later Penrose [63] have been quite forceful and convincing in their rejection of the strong AI position. Searle noted ironically that, just because we can simulate a rainstorm does not imply that "a computer simulation of a rainstorm will leave us all drenched". "Why on earth", he continues, "would anyone suppose that a computer simulation of understanding actually understood anything?". Penrose, a renowned British physicist and mathematician, however, went further, by arguing that modern day computers, as limited incarnations of Turing machines, can not *even* imitate humans. He suggested that for a machine to ever be able to match human abilities, we must go beyond the present day von Neumann architecture, perhaps, by exploiting quantum effects or other physical properties yet to be discovered.

As these deeply philosophical questions are, quite often, a matter of belief rather than hard proof, let us now state our own position. At the root of our perspective is the axiomatic belief that every self-aware individual can be sure of the existence of only two things, both

[3]Based on the work of such scientists as Newell, Simon, Schank, Winograd, etc. in the 1960s and 1970s.

[4]Sometimes even called "cautious" AI.

of which can be experienced directly: the realm which is under the direct control of this individual (the "self") and the realm which is beyond his direct control (the rest of the world, or the "universe"). Relative to any such self-aware individual, all other human beings "exist" only in the second realm of the "universe". They exist only to the extent that the "self" is willing to attribute such an "existence" to them. Here, the Cartesian axiom "I think, therefore I am" applies to "self" only, and does not apply to others. "They appear to be thinking (or understanding, or feeling, etc.), therefore they are, or, therefore, they are exactly like me" can only be a theory to forever remain unproven. It would be safe to say, however, that most individuals do subscribe to this theory, and it is precisely this theory that "glues" the individual "selves" together into the society or mankind, as we know it.

Since it is doubtful that any cognitive processes perceived in the realm of "self" can ever be proven to apply to any human or artificial system outside of "self", all attempts to construct an artificial system that duplicates "self" must be futile. What should be the goal of the AI research then? It is our position that the best we can ever hope for, is to construct systems that are resigned to the imitation of the capabilities of "other" humans or animals based on the observed behaviours outside of "self". This is essentially a behaviouristic stance that was much criticized by Searle:

> "The Turing test is typical of the tradition in being unashamedly behaviouristic and operationalistic, and I believe that if AI workers totally repudiated behaviourism and operationalism much of the confusion between simulation and duplication would be eliminated."

Since we believe that the duplication of "self" is impossible anyways, all we have left is simulation.

Out of all observable human behaviours, the AI research must be concerned with imitating "intelligent" or rational behaviours. We must cast away the romanticized notions of building a machine that *appears* to love or hate, or experience any other emotions for that matter.

In fact, we should not be interested in imitating any aspects of human behaviour except for the rational ones. After all, we *already know* how to "build" human beings, that can play, love, or write poetry. We must, instead, solely concentrate on building machines that simulate, or imitate, and even surpass the human capabilities of rational decision making. A machine successfully imitating such capabilities should be equivalent to having a human decision maker employed to solve a given task.

While it may be difficult to imagine, today, how such a simulation can ever resemble the human level of "intelligence", given the state-of-the-art at the moment, there exist already examples of technological advances in machine intelligence that would be quite inspirational. A small spider robot can learn to move in a given direction after interacting with its environment, starting with random jerks and motions of the legs and leading to an efficient straight-line motion in a matter of minutes. To an observer, unaware of the learning algorithms involved, such a behaviour may appear to possess exactly the properties of intentionality and purpose, as sought by Searle.

Based on the above line of thought, we believe that any successful theory of intelligence must be based on the observable behaviours of the systems to which such intelligence is attributed. Sternberg, the giant of psychological intelligence research, notes in [80, p.8]:

> "Many theorists of intelligence would define the locus of intelligence as occurring neither wholly within the individual nor wholly within the environment, but rather within the interaction between the two ... Thus it may be difficult to understand intelligence fully without first considering the interaction of the person with one or more environments ..."

In the most general case, we cannot assume that the entity being evaluated is willing to communicate with us in the sense of the Turing test or, in general, is willing to cooperate in such an evaluation. Intelligent behaviour, we believe, is necessarily goal-oriented and must involve the interdependent abilities to learn (acquire the rules of the environment) and

plan (apply these rules to reach its objectives). Since any behaviour occurs in the context of interaction with the surrounding environment, any intelligent behaviour must be studied in exactly the same setting. This is precisely the inspiration for our generalization of the Agent-Environment interaction protocol (Æip) offered in Chapter 3 (which is the central building block of the thesis), and all our subsequent results.

Appendix B

Adaptive Data Structures

This Appendix complements the review of Adaptive Data Structures included in Chapter 5. It closely follows a recent review performed by Abdelrahman Amer [6].

B.1 Asymptotic Cost

While the self-organizing data structure has no knowledge of the probabilities that produce the observed sequence of queries, it is generally assumed that the sequence is based on a non-uniform distribution where some records have higher access probabilities than others. It is shown in [33], that without such an assumption no amount of reordering can result in an improvement in access time.

With this assumption, however, and *if* the algorithm can be expected to converge over time, the average search cost attains its asymptotic value. The term "average search cost", therefore, usually means the average search cost *after* convergence, and consequently, the phrases "average", "asymptotic", and "expected search cost" can be used interchangeably.

The expected search cost for algorithm A is given by the expected value law over all list

configurations:

$$\begin{aligned} ExpectedCost(A) &= \sum_{1\leq j\leq n!} (P\{\pi_j\}_A \; cost(\pi_j)) \\ &= \sum_{1\leq j\leq n!} (P\{\pi_j\}_A \sum_{1\leq i\leq n} s_i\pi_j(i)), \end{aligned} \tag{B.1}$$

where $cost(\pi_j)$ is given by Equation (5.1). In practice, however, it is often difficult to derive the probabilities of convergence of A to all possible asymptotic list configurations. Instead, several authors (e.g. [32, 68]) have shown another way to compute the expected search cost that depends on the probability that one record precedes another.

It has also been noted since McCabe's work [45] in 1965 that self-organizing lists can be analyzed through the theory of Markov chains. A list with n records can assume any of $n!$ configurations by applying the permutation algorithm (scheme) A. Each permutation results in a switch from one configuration (one state of the Markov chain) to another configuration (another state of the Markov chain). This can be represented by a Markov chain with the transition matrix of $n!$ states, where every state represents a particular configuration $(\pi_j, 1 \leq j \leq n!)$ of the list. Transition from one state to another occurs by applying the permutation algorithm, and is therefore dependent on the access probabilities. A simple example of Markov chain based analysis can be found in [26].

For the purposes of analysis, we can also exploit the time reversibility property of some Markov chains. A chain is *time reversible* when, starting from a given state, the probability of following a loop back to that state is the same as the probability of the reverse path. The reader can find a thorough analysis of time reversible algorithms in [26]. Time reversibility is a powerful analysis tool because of the following property: for a time-reversible self-organizing list with a transition matrix M,

$$P\{\pi_i\}m_{i,j} = P\{\pi_j\}m_{j,i} \tag{B.2}$$

for all $i \neq j$. This property was first described by Rivest [68] in his analysis of the transposition rule, and was later generalized by Oommen and Dong [59] in their analysis of the

swap-with-parent scheme. It gives a way by which we can find the relative probabilities of two different list configurations in terms of transition probabilities.

B.2 Amortized Cost

In the amortized analysis [17, 79] of a self-organizing data structure, we consider the cost averaged over *all* operations performed, rather than just after convergence. Some researchers have used amortized analysis to consider the worst case sequence of requests. Bentley and McGeoch [17] have provided empirical results that suggest that the amortized analysis of permutation heuristics better describes the behaviour of the heuristic on real data, than a probabilistic analysis.

The amortized cost analysis is especially important in non-stationary environments because there is no fixed optimal action over time. Narendra and Thathachar [51] define the optimality of a system in a non-stationary environment if it minimizes

$$\lim_{T\to\infty} \frac{1}{T} \sum_{n=1}^{T} E[\beta(n)],$$

where $E[\beta(n)]$ is the expected cost at time n, which is exactly the definition of amortized cost.

B.3 Relative Measures and Competitiveness

The performance of self-organizing lists can also be measured by comparing the specific algorithm to another, possibly hypothetical, algorithm for the same request distribution. For instance, we can compare the performance of an algorithm to that of list with the *optimal static ordering*. In other words, the self-organizing list is compared against another list whose records are arranged in the descending order of the static access probabilities [2]. Curiously, the optimal static ordering may not always be the ordering that would minimize the search

cost. Consider sequences that are characterized with *locality of reference*. Subsequences of these can have access frequencies that are different from the overall access frequencies. In such a setting, the optimal static ordering may not be "optimal".

Competitiveness is another relative measurement of an online algorithm against an offline one. An *online* algorithm does not know the access sequence beforehand, whereas in an *offline* setting, the entire sequence is known *a priori* and the algorithm can serve the requests at a minimum cost. An online algorithm is called *c-competitive* if, for any request sequence, its cost is no more than c times the cost of the optimum offline algorithm for that sequence [67]. Sleator and Tarjan [79] were the first to use competitive analysis in their analysis of the MTF algorithm.

While reading the remainder of this review, we would like the reader to note (so as to avoid confusion) that the notion of c-competitiveness and the notion of a *competitive setting*, within which we conduct our experiments on algorithm comparison, are unrelated.

B.4 Convergence

In addition to the cost analysis, we are also interested in convergence properties of algorithms, which, unfortunately, are not nearly as well studied as the different types of cost. Bitner [20] suggested a measure of convergence that he called *overwork*. The overwork of an algorithm is defined as the area between the cost curve and its asymptotic level. A steep drop in the cost curve indicates that the algorithm is converging fast, and this will result in a smaller overwork area. Therefore, the smaller the overwork, the faster the convergence. To compare two algorithms, Bitner suggested measuring the area between their respective costs. This technique comes in handy when one algorithm offers fast convergence to a suboptimal cost, while another converges slowly to a nearly optimal level of asymptotic cost (see our empirical comparison of MTF and TR in a traditional setting in Section 5.3.3).

Except for some specific distributions and a limited number of algorithms, a closed form

expression for the overwork is very difficult to obtain. Instead of an analytical solution, researchers often have to choose empirical methods. If an algorithm is allowed to run for a sufficiently long time, the final cost level is then taken as an approximation of the true asymptotic cost. The time for convergence would then be the amount of time taken for the measured cost to come within an arbitrary percentage of the asymptotic cost. We also take the empirical approach in our comparative study of algorithms in Section 5.3. While convergence of self-organizing algorithms does not seem to be of great concern throughout the literature, the speed of convergence proves key to the success of algorithms in a competitive setting.

B.5 The Move-To-Front (MTF) Rule

The average search cost analysis of the move-to-front heuristic can be found in [20, 37, 45, 68]. In order to calculate the average search cost we need a closed form expression for the probability that record R_j precedes R_i for a given rule. Here is a simple derivation of this expression taken from [32, 68]: R_j will precede R_i if there has been a request for R_j subsequent to the most recent request of R_i. Let the time elapsed since that request be k. If R_j precedes R_i, we know that k requests ago there had been one request to R_j, and since then, $k-1$ requests for records other than R_i or R_j. If we sum the probability of this event occurring over all k, we get:

$$\begin{aligned} P\{R_j \text{ precedes } R_i\} &= s_j * \sum_{1 \le k \le \infty} (1 - s_i - s_j)^{(k-1)} \\ &= \frac{s_j}{s_i + s_j}. \end{aligned} \tag{B.3}$$

Based on the methodology of [32, 68], we obtain the average cost of the MTF rule (A_{MTF})

to be:

$$\begin{aligned} A_{MTF} &= 1 + \sum_{1 \le i \le n} \sum_{j \neq i} (s_i \, P\{R_j \text{ precedes } R_i\}) \\ &= 1 + \sum_{1 \le i < j \le n} \sum (s_i \, P\{R_j \text{ precedes } R_i\} + s_j \, P\{R_i \text{ precedes } R_j\}) \\ &= 1 + \sum_{1 \le i < j \le n} \sum (\frac{s_i s_j}{s_i + s_j} + \frac{s_j s_i}{s_i + s_j}) \\ &= 1 + 2 \sum_{1 \le i < j \le n} \sum \frac{s_i s_j}{s_i + s_j}. \end{aligned} \tag{B.4}$$

Compared to the optimal static ordering, Rivest [68, illustrated in [26]] shows how the MTF rule does not do more than twice the average optimal work, which has the expression:

$$A_{STAT} = \sum_{1 \le i \le n} (s_i * i). \tag{B.5}$$

Taking the ratio (B.4) to (B.5), we get:

$$\begin{aligned} \frac{A_{MTF}}{A_{STAT}} &= \frac{1 + 2 \sum \sum_{1 \le i < j \le n} \frac{s_i s_j}{s_i + s_j}}{\sum_{1 \le i \le n} (s_i * i)} \\ &= \frac{1 + 2 \sum_{1 \le i \le n} (s_i * \sum_{1 \le j < i} \frac{s_j}{s_i + s_j})}{1 + \sum_{1 \le i \le n} (s_i * (i - 1))}. \end{aligned} \tag{B.6}$$

Notice that since $s_i \ge 0$,

$$\sum_{1 \le j < i} \frac{s_j}{s_i + s_j} \le \sum_{1 \le j < i} \frac{s_j}{s_j} = \sum_{1 \le j < i} 1 = i - 1,$$

and therefore,

$$\begin{aligned} \frac{A_{MTF}}{A_{STAT}} &\le \frac{1 + 2 \sum_{1 \le i \le n} (s_i * (i - 1))}{1 + \sum_{1 \le i \le n} (s_i * (i - 1))} \\ &= \frac{1 + 2x}{1 + x} \\ &= 2 \, (1 - \frac{1}{1 + x}) \\ &\le 2. \end{aligned}$$

Chung *et al* [23] later proved that $\frac{A_{MTF}}{A_{STAT}} \leq \pi/2 \approx 1.5708$ for any distribution. Bentley and McGeoch [17] proved that the amortized search cost for the MTF is at most twice that of the optimal static ordering if the list initially contains the items ordered by first access.

Sleator and Tarjan [79] also proved that the MTF is 2-competitive. They proved that, for any algorithm A (online or not), and any access sequence S, if reordering costs are not counted, then:

$$C_{MTF}(S) \leq 2C_A(S) - m,$$

where C is the total cost of all operations, and m is the number of accesses. Their proof utilized the concept of a *potential function*, which maps the configuration of a list to a real number, Φ. If an operation at time t costs $c(t)$, and caused the list to shift from potential $\Phi(t-1)$ to $\Phi(t)$, they defined the *amortized time* of such an operation to be $c(t)+\Phi(t)-\Phi(t-1)$, i.e. the running time plus the change in the potential. Both algorithms MTF and A were run in parallel on two lists that were initially identical.

Let us define the potential function to be the number of inversions in the MTF's list with respect to A's list, where an inversion in one list with respect to the other is defined as an unordered pair i and j such that i occurs anywhere before j in one list and anywhere after j in another. Clearly, since the lists are initially identical, the initial potential is zero.

Consider an access on element x. Let k denote the number of elements that precede x in both MTF's and A's lists. Further, let l denote the number of items that precede x in MTF's list but not in A's. In such a case, $c_{MTF}(t) = k + l + 1$ and $c_A \geq k + 1$. When MTF serves the request and moves x to the front of the list, l inversions are destroyed, and simultaneously, at most k new inversions are created. Thus, the amortized time $a_{MTF}(t)$ is:

$$\begin{aligned} a_{MTF}(t) &= c_{MTF}(t) + \Phi(t) - \Phi(t-1) \\ &\leq c_{MTF}(t) + k - l = 2k + 1 \\ &\leq 2c_A(t) - 1. \end{aligned}$$

Summing over all t, we get:

$$\begin{aligned} \sum_{1 \le t \le m} a_{MTF}(t) &= C_{MTF} + \Phi(m) - \Phi(0) \\ &\le 2C_A - m. \end{aligned}$$

Since the lists are initially equal, the initial potential $\Phi(0)$ is zero and the final potential $\Phi(m)$ is nonnegative. Therefore, $C_{MTF} \le C_{MTF} + \Phi(m) - \Phi(0) \le 2C_A - m$.

This is a powerful result because it implies that, for *any* sequence of requests, the MTF algorithm is bounded by being within a constant factor as efficient as *any* algorithm that moves the accessed element closer to the front of the list. This includes algorithms with lookahead, and even the ones with knowledge of the entire access sequence, because the *a priori* knowledge of access sequences cannot significantly reduce the cost. Karp and Raghavan [67, reported personal communication] later suggested that the competitive ratio of 2 is the best ratio that a deterministic online algorithm, for the list update problem, can achieve.

According to Amer [6], this theorem does not cover algorithms that do not move accessed elements closer to the front of the list (e.g. move-to-rear algorithm), because the l inversions might not be destroyed and the k inversions might not be created. For the same reason, algorithms that move more than one element are also believed to not be covered by this theorem.

B.6 The Transposition (TR) Rule

In contrast to MTF, the transposition rule is a time-reversible algorithm, as the original list can be reconstructed by reversing the access sequence. Equation (B.2) gives a relation between two specific configurations of a time-reversible system in terms of transition probabilities. Rivest [68] offers a more specific version of this relation for the transposition rule.

According to Rivest, the system's stationary probabilities obey the following relation:

$$\frac{P\{\pi_a = < R_{i_1} R_{i_2} \ldots R_{i_j} R_{i_{j+1}} \ldots R_{i_n} >\}}{P\{\pi_b = < R_{i_1} R_{i_2} \ldots R_{i_{j+1}} R_{i_j} \ldots R_{i_n} >\}} = \frac{s_{i_j}}{s_{i_{j+1}}}, \tag{B.7}$$

for $1 \leq j < n$ if $s_k \neq 0$ for $1 \leq k \leq n$. In other words, the ratio of the probability of a list configuration before and after a pair of elements are transposed is the same as the ratio of the respective access probabilities of these elements. Observe that the following relation gives the ratio of the probabilities of any two list configurations π_a and π_b:

$$\frac{P\{\pi_a\}}{P\{\pi_b\}} = \frac{P\{\pi_a\}}{P\{\pi_{i_1}\}} \times \frac{P\{\pi_{i_1}\}}{P\{\pi_{i_2}\}} \times \ldots \times \frac{P\{\pi_{i_{k-1}}\}}{P\{\pi_{i_k}\}} \times \frac{P\{\pi_{i_k}\}}{P\{\pi_b\}},$$

for some $1 \leq k \leq n! - 2$. Using Equation (B.7), this relation can be rewritten as:

$$\frac{P\{\pi_a\}}{P\{\pi_b\}} = \prod_{u,v} \left(\frac{s_u}{s_v}\right), \tag{B.8}$$

for some $1 \leq u, v \leq n$. Let $\delta_i(\pi_a, \pi_b)$ denote the number of places that R_i is displaced from its position during transformation from π_a to π_b. Further, let this quantity be positive if R_i is moved further from the list head (i.e. R_i appeared in π_a in a position prior to its position in π_b) and negative if R_i is moved closer to the list head. For every transposition of R_i, s_i will appear in the numerator of Equation (B.8) when it moves down the list and in the denominator when it moves up the list. Thus, all instances of the term s_i in Equation (B.8) can be grouped as $s_i^{\delta_i(\pi_a,\pi_b)}$, and the equation can be rewritten as:

$$\frac{P\{\pi_a\}}{P\{\pi_b\}} = \prod_{1 \leq i \leq n} s_i^{\delta_i(\pi_a,\pi_b)}. \tag{B.9}$$

Now that we can find an expression for the probability of any list configuration π in terms of the probability of the initial configuration π_0, we can use Equation (B.1) to find the expected search cost under the transposition rule A_{TR} as given by Rivest to be:

$$A_{TR} = P\{\pi_0\} \sum_{\pi} \left(\left(\prod_{1 \leq i \leq n} s_i^{\delta_i(\pi_0,\pi)} \right) \sum_{1 \leq j \leq n} s_j \pi(j) \right),$$

where

$$P\{\pi_0\} = \left(\sum_{\pi} \prod_{1 \le i \le n} s_i^{\delta_i(\pi_0,\pi)} \right)^{-1}.$$

The details of the derivation are omitted and can be found in [26].

Rivest [68] proved that except for uninteresting lists[1], the transposition heuristic always has a lower asymptotic cost than that of the MTF. This is a rather important result for our present study, as it forms the basis for a widely supported conclusion that TR is superior to MTF in a traditional setting. The present study shows, that in contrast to the traditional setting, MTF is a *superior scheme* when a list is allowed to be perturbed by an external order-destroying force.

Rivest conjectured that TR minimizes the average search cost for any distribution, but Anderson *et al* [7] found a counterexample. Rivest also performed an empirical analysis on the TR, and more specifically concentrated on requests that obeyed Zipf's law. On these sequences, Rivest found that TR performs better than MTF. In contrast, Bentley and McGeoch [17] ran their experiment on requests from text and Pascal files, and found that TR performed worse than both MTF and FC. Yet, they reported that TR outperformed the expected results of their analysis, and that the output may have been different had Pascal's reserved words been treated differently than identifiers.

Dong [26] showed empirically that the transposition rule yields an asymptotic search cost that is always better than that of MTF for five different probability distributions. She performed 12 parallel experiments on lists with 100 records for each of the following distributions: Zipf, 80-20, Lotka, exponential and linear. She first generated 250,000 queries on the lists to allow the algorithms to converge. She then averaged the cost over the last 50,000 queries (time average) and over the twelve experiments (ensemble average), to get an estimate of the asymptotic cost.

All of the above leads us to the comparison of TR, MTF, and the optimal static ordering,

[1]Lists with fewer than three records or those where all nonzero probabilities s_i are equal.

which we can summarize as a relation that holds true, for most sequences that obey a stationary distribution:

$$A_{TR} \leq A_{MTF} \leq \frac{\pi}{2} * A_{STAT}.$$

By providing a counterexample, Bentley and McGeoch [17] showed that, unlike MTF and FC, TR does not have the property of a constant-bound amortized cost, compared to the amortized cost of the optimal static ordering. Consider the access sequence $ABCDE(ED)^k$. In TR, the last two elements will keep swapping positions and the average search cost will be around 5, whereas it is 1.5 in the optimal ordering.

Sleator and Tarjan [79] concerned themselves with the competitiveness analysis and showed that TR is not c-competitive for any constant c that is independent of the list size n, citing the same counterexample by Bentley and McGeoch.

It is also an accepted fact in the literature that TR is the slowest list organizing strategy. Using his definition of the overwork measure of convergence, Bitner [20] shows that for a particular distribution, where $s_1 = 0$ and $s_i = 1/(n-1), 2 \leq i \leq n$, the overwork for the transposition rule is $(n^2 - 1)/6$ and for the move-to-front rule is $(n-1)/2$. Thus, there is an *order of magnitude* of difference between the two. This result is in line with our own comparison of MTF and TR in a competitive setting, suggesting that the overwork measure is correlated with our own metric implicit in the competitive nature of comparison.

Bitner suggested another measure for comparing the convergence of two algorithms by counting the *total expected cost*, which is the sum of the costs for r accesses. From his results, it is noticeable that the total expected cost for the transposition rule over Zipf distribution grows quadratically with the number of accesses, indicating poor convergence prospects.

In the experiments cited above, Dong also confirmed empirically that the transposition rule always converges to a steady state much slower than the MTF does.

Appendix C

On Generating Equiprobable Patterns

In Chapter 5, we defined a "pattern" as a probability distribution over n queries corresponding to each element of the list. We would like to choose these patterns at random, ensuring that every possible pattern is drawn in an equally likely manner. This is what we call an *equiprobable pattern.* In generating such an equiprobable pattern, we must divide the unity probability among the n elements of the list, without favouring patterns of a particular "shape"[1]. Before we proceed, we demonstrate, first of all, that the two very straightforward strategies for generating such patterns fall short of our standard of uniformity. In generating such pseudo-random equiprobable patterns, we are going to assume, as usual, that we have access to a function u that generates a sequence $\{u_i\}$ of pseudo-random floating-point numbers uniformly distributed in the interval $[0, 1]$. Our aim is to produce a new pattern described by a random vector $\vec{p} = [p_1, p_2, \ldots, p_n]^T$, where p_i is the probability of the i^{th} element of the list queried and n is the number of elements in the list.

When $n = 2$, it is rather obvious how to generate the corresponding equiprobable pattern with a single random number u_1:

$$p_1 = u_1 \quad \text{and} \quad p_2 = 1 - p_1. \tag{C.1}$$

[1]By *shape* of a pattern, here, we mean the shape of the resulting query distribution or histogram.

The entire pattern is defined by a single random variable which, by definition of the u function, is uniformly distributed and, hence, equiprobable. Surprisingly, it is much less obvious how this can be done for $n > 2$.

Consider the simple scheme which assigns

$$\begin{aligned} p_1 &= u_1 \\ p_2 &= u_2(1 - p_1) \\ p_3 &= u_3(1 - p_1 - p_2) \\ &\cdots \\ p_{n-1} &= u_{n-1}(1 - \textstyle\sum_1^{n-2} p_i) \\ p_n &= 1 - \textstyle\sum_1^{n-1} p_i. \end{aligned} \tag{C.2}$$

Clearly, scheme (C.1) is a special case of scheme (C.2). The problem, however, is that under such a procedure the expected value of p_1 is 0.5 or 2^{-1}. In general, for $i = 1, 2, \ldots, n-1$

$$E[p_i] = E[u_i(1 - \sum_{k=1}^{i-1} p_k)] = E[u_i](1 - \sum_{k=1}^{i-1} E[p_k]) = \frac{1}{2}(1 - \sum_{k=1}^{i-1} E[p_k]).$$

The above Equation relies on the fact that u_i are independent random variables. Solving the simple recurrence relation, we get

$$E[p_i] = \frac{1}{2^i}.$$

Patterns with such a property clearly do *not* match our criteria for being equiprobable because patterns of a particular shape[2] are being preferred. In fact, by a symmetrical argument, a correct generating procedure should have the same expected query probability for all list elements, specifically equal to $\frac{1}{n}$. This procedure satisfies this requirement only for a special case of $n = 2$ (i.e. two-element list), but such a simple case is of little practical interest.

[2]In this procedure the most probable element is expected to be 2^{n-2} times more likely as compared to the least probable.

Another simple scheme consists of assigning

$$p_i = \frac{u_i}{\sum_{j=1}^{n} u_j}. \tag{C.3}$$

In other words, we ensure that the probabilities add up to unity through normalization. This procedure clearly does satisfy the "equality of expectation" requirement mentioned earlier, but, unfortunately, still falls short of the ultimate goal of producing equiprobable patterns. This latter point may be rather less obvious. We know that to be equiprobable, scheme (C.3) must behave identically for $n = 2$ to scheme (C.1), as the latter was trivially equiprobable. Unfortunately this is not the case. In order to demonstrate this we can compute the probability $\Pr[p_1 < \frac{p_2}{2}]$ for each of the schemes:

$$\text{Scheme (C.1)}: \quad \Pr\left[p_1 < \frac{p_2}{2}\right] = \Pr\left[u_1 < \frac{1}{2}(1-u_1)\right] = \Pr\left[u_1 < \frac{1}{3}\right] = \frac{1}{3},$$

$$\text{Scheme (C.3)}: \quad \Pr\left[p_1 < \frac{p_2}{2}\right] = \Pr\left[\frac{u_1}{u_1+u_2} < \frac{u_2}{2(u_1+u_2)}\right] = \Pr\left[u_1 < \frac{u_2}{2}\right] = \frac{1}{4}.$$

The above clearly demonstrates that scheme (C.3) is not equiprobable.

The procedure we describe next, generates equiprobable patterns. This assertion is based on an argument which can easily be formalized, involving combinatorics and approximation of real valued probability distributions by discrete valued ones. We begin by restating a well known problem in combinatorics.

Suppose four friends sit down to enjoy a pizza that was cut into 12 identical slices. The easiest and possibly most fair way to divide this pizza among the friends is to give three slices to each one. But this is obviously not the only possibility. Assuming we allow a single person to receive any number of slices from 0 to 12, as well as allow any number of slices to remain uneaten, how many ways of sharing this pizza are there? The solution to this problem is a little harder than it looks at first.

The objective is clearly to divide the 12 slices into 5 distinct and possibly empty groups of slices, one for each friend and the additional "leftover" group. We can visualize a particular

partitioning of slices Δ into these 5 groups by introducing group separator markers '|':

$$\Delta \mid \Delta\ \Delta\ \Delta\ \Delta \mid \mid \Delta\ \Delta\ \Delta \mid \Delta\ \Delta\ \Delta\ \Delta.$$

As we can see, an empty group is shown as two adjacent '|' markers. It is now easy to see that to choose a partitioning we simply need to choose the positions of the four separator markers, i.e. choose four positions out of the $12 + 4 = 16$ possible positions. There exist, thus,

$$\binom{16}{4} = \frac{16!}{4!12!} = 1820$$

possible partitionings in total.

This problem is a discrete version of the problem of generating equiprobable distributions. The number of groups corresponds to the number of list elements n, while the total number of slices is our discretization parameter which subdivides the total probability weight of unity into m identical portions. As we move to the continuous probability space, $m \rightarrow \infty$, while the number of separator markers stays the same, namely $n-1$. The problem of choosing an equiprobable distribution has, thus, been reduced to choosing $n-1$ random marker positions on the interval $[0,1]$, or, simply, $\{u_1, u_2, \ldots, u_{n-1}\}$. This is exactly the procedure described in Section 5.2. After sorting the markers $\{u_i\}$ in an ascending order, we obtain $\{u_{s_j}\}$, where $\{s_j\}, j = 1, 2, \ldots, n-1$ are the indices i in the sorted order. All that remains is to assign

$$\begin{aligned} p_1 &= u_{s_1} \\ p_k &= u_{s_k} - u_{s_{k-1}} \quad k = 2, 3, ..., n-1 \\ p_n &= 1 - u_{s_{n-1}}. \end{aligned}$$

Appendix D

JAGUAR Experimentation Tool

The vast majority of the experiments presented in this thesis were conducted using the tool we called JAGUAR. JAGUAR is an acronym for *Java AGents Unified ARchitecture* and is a GUI-based Object-Oriented tool which allows the user to conduct *Interaction Learning* experiments. Since JAGUAR is a concrete implementation of the Æip framework, an experiment consists of a collection of interconnected *entities* communicating with each other. The remainder of this appendix describes JAGUAR's architecture, API and examples of its usage.

D.1 JAGUAR Architecture

JAGUAR contains a core non-GUI library which defines the components required for setting up Interaction Learning experiments. In the interest of brevity, we mention only the most important classes and interfaces, and omit the rest. The Java interface `EntityInterface` defines the standard behaviour of all entities. Each entity is defined to be a source of `StateChangeEvents` used by the Interaction Manager. Besides the expected accessor methods, the entity can be notified when communication lines are being connected to its ports

through the methods `connectLineToInport()` and `connectLineToOutport()`. Each entity also includes the following experiment life cycle methods:

- `start()` is called by the Interaction Manager at the start of each episode. In non-episodic tasks it is called once at the beginning of the experiment.
- `tick()` is called by the Interaction Manager to advance the clock of the entity. In discrete-time interactions it initiates a new sense/act/reinforce interaction cycle.
- `exit()` is called by the Interaction Manager at the end of the experiment.

The Java interface `LineInterface` specifies how abstract communication lines work. Data propagates through a line in one direction only, and the current value available from the line is accessible through the `getState()` method. Lines connect their end points to the entities using the `connectToSource()` and `connectToSink()` methods.

The Java class `Experiment` is the key class that describes how an experiment is put together. It defines:

- methods for the assembly of entities into an experiment, e.g. `newEntity()`
- methods for connecting outports to inports using communication lines, e.g. `newLine()` or `autoConnect()`
- methods for specifying other simulation parameters (such as how the entities' actions are to be interleaved in time), e.g. `setTicksToSkip()`, and
- methods for advancing the simulation clock, whether by individual clock *ticks* (as in `clockTick()`) or by complete *episodes* (as in `episode()`).

Besides the types that define the framework of JAGUAR, we have also compiled a library of pre-programmed entity classes that can be configured to suit each new experimental setup.

Once a general-purpose learning algorithm is implemented, it can be reused repeatedly in a variety of experiments. This is one of the greatest advantages of JAGUAR.

Entities are usually characterized as being either adaptive (Agents) or not (Worlds). Some Agent entities may also be designed such that some of the outport signals (i.e. some aspect of the observable state of the Agent) behave in a non-adaptive fashion. In such cases the subdivision into Agents and Worlds may not be so straightforward, especially in experiments where one entity acts as an (albeit changing) Environment/World for the other. Additionally, a number of entities are created solely to help connect the entities or to adapt the existing entities to new experimental conditions.

We catalogue below some of the entities that were implemented in JAGUAR. All of these entities (except for the `PolicyIterationAgent`) were used in the experiments reported in this thesis:

- World Entities
  - `GridWorld`
  - `TicTacToeWorld`
  - `LightsOutWorld`
  - `ListWorld` — for experiments with Adaptive Data Structures
- Agent Entities
  - `PolicyInterationAgent` — was used to explore RL methods based on dynamic programming
  - `QLearningAgent`
  - `DQLearningAgent`
  - `Agent` — for experiments with Adaptive Data Structures

- Support Entities
  - `ArraySplitter` — splits a composite signal into individual components
  - `SymmetryMap` — maps several symmetrically equivalent states of LightsOut puzzle to a single state
  - `Controller` — combines actions of two Agents into a single robot action

It should be highlighted that all of the entities are fully configurable. The dimensions of the grid in the `GridWorld` entity, for instance, can be specified at the beginning of an experiment through a configuration file. Likewise, the dimensions of the Q table can be specified for the `QLearningAgent`, while the arity of the task can additionally be provided for the `DQLearningAgent`. In general, each entity defines a number of configurable properties that experimenters can modify before the outset of an experiment, in order to adapt the entity to a new experiment.

Given the core classes and the collection of library entities one can already program and run experiments. While this approach alone presents great flexibility to the experimenter, we would like to have an easy-to-use graphical interface to allow the setup of reasonably complex experiments without the need to write a single line of code. To satisfy this desire, a simple graphical user interface was constructed as a layer on top of the core JAGUAR classes. The following section demonstrates, step by step, how an experiment can be conducted using the JAGUAR GUI.

D.2 An Example of Experimental Setup

Figure D.1 shows that JAGUAR starts with a "empty" experiment. At this point we can load a previously saved experiment and continue the simulation, or set up a new experiment by either creating new entities, or by loading previously saved entities and then connecting them with communication lines. The New Entity dialog box shows that an entity is created by

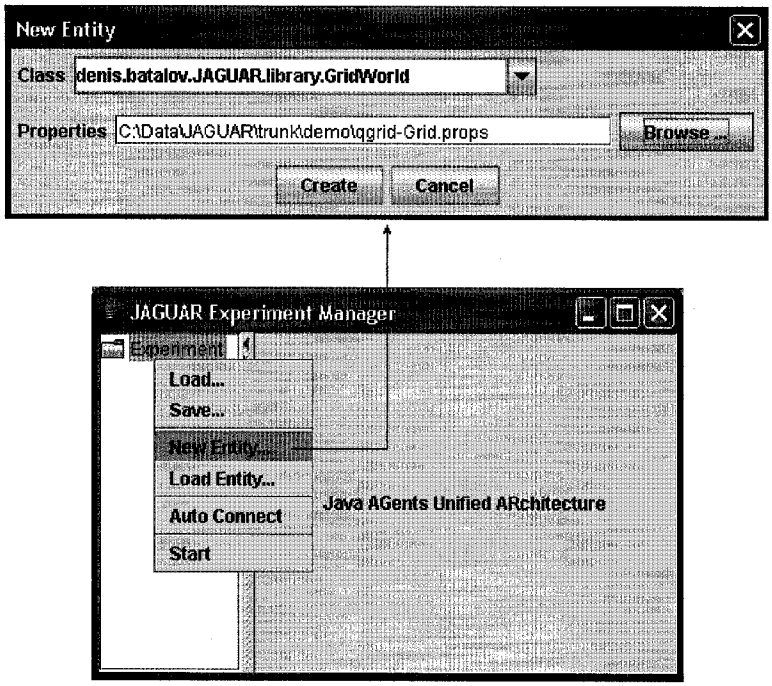

Figure D.1: Adding a brand new entity to a fresh experiment.

choosing the entity class from a drop down box, and by additionally selecting the *properties* file, which is essentially a text file configuring the entity.

By way of example, the configuration file used for the `GridWorld` entity consists of the following statements:

```
entity.grid.numRows    = 10
entity.grid.numColumns = 10

entity.robot.initRow    = random
entity.robot.initColumn = random

entity.goalCell.rows    = 0
```

```
entity.goalCell.columns = 0

entity.reinforcement=1 # select the reinforcement function for outport 0

entity.eoe.moveLimit=18

entity.log.pathname=gw-default.txt # path to the log file
entity.log.comm.interval=10 # add a log record every Nth episode

entity.algorithm.randomSeed=12345

entity.visualizers=denis.batalov.JAGUAR.library.GridWorldVisualizer
```

In this file we specify the dimensions of the grid, the initial position of the robot (random in this case) and the location of the goal cell which the robot must learn to reach. The `reinforcement` property specifies the reinforcement schedule to be emitted from the reinforcement outport of the `GridWorld` entity. The `moveLimit` property indicates the number of robot moves that would cause the episode to end. In an $m \times n$ rectangular grid the robot need not take more than $m + n - 2$ moves (in this case $10 + 10 - 2 = 18$) and so we terminate the episode after this period of time. The `visualizers` property specifies the list of Java classes that implement different kinds of *visualizers* for this entity, where a visualizer is essentially a stand-alone window which graphically displays the current state of the entity. Figure D.2 depicts a typical visualizer.

For the purposes of this step-by-step demonstration we will set up a simple experiment where a robot controlled by a Q-learning agent learns to move the robot to the goal cell. From the current cell the robot can move to any of the four immediately adjacent cells. If the robot is next to the wall and attempts to move in the direction of the wall it will remain in the same cell, essentially wasting time.

Figure D.2 shows that in addition to the `GridWorld` entity, the `QLearningAgent` had also been created. Both entities are expanded in a tree view to show all the inports and outports. Note that the End of the Episode (labeled `EOE`) outport is shown in bold. That is because this outport was designated through the GUI as the one responsible for signaling the end of the episode to the experimentation system. Individual ports can be connected to one another through the pop-up menu, but an `Auto Connect` option in Figure D.1 can also be chosen. If the ports of different entities were named to match each other, they can be automatically connected, offering a great deal of convenience to the user.

The peculiarly named `Tick Skipper` panel allows the experimenter to control how the entities are to interleave their actions in time. Since in this experiment we do not want the agent and the grid to act simultaneously, we indicate that each will skip every other clock tick. In order to indicate which of the entities will start acting, we specifying how many ticks have already been skipped prior to the start of the episode. We want the agent to act before the grid and so we indicate that the agent has already skipped one tick. For the grid entity we specify the number of skipped ticks as 0, so as to offset it in time. While this may not be the most intuitive way of specifying such interleaving, it afforded us enough flexibility coupled with the simplicity of UI implementation. We expect future versions of the software to provide a more intuitive interface.

Figure D.2 also shows the visualizers for each of the entities opened in separate windows. For the `GridWorld` entity, the corresponding visualizer shows the location of the goal cell (shaded) as well as the location of the robot (circle). The visualizer for the `QLearningAgent` is a crude depiction of the robot (view from above) showing the two wheels and the direction in which the robot is to move next. Note that, since the `QLearningAgent` is a general-purpose agent, not specifically designed to interact with the `GridWorld`, a custom robot-in-grid visualizer had to be implemented. The same goes for the text-based policy view (a matrix of crosses).

Once the entities are connected and their temporal interleaving is established, we are

ready to start the experiment. After the start of the experiment no part of the entity or experiment configuration is allowed to change. Figure D.3 displays the experiment control panel which allows the experimenter to advance the clock either by individual ticks or by entire episodes. The figure shows that the experiment was advanced by a 1000 episodes. The policy view in the panel that corresponds to `QLearningAgent` clearly shows that the agent is very close to converging to the optimal behaviour. The box-drawing ASCII characters were used to indicate the best discovered actions for each cell. The double-base T-shaped symbols are meant to represent arrows pointing towards the goal state. The cross that remains in the goal cell indicates that all four directions are equally appropriate from the goal state because the episode is terminated upon reaching the goal, and that no subsequent state exists. The optimal behaviour still needs to be learned in two cells only. In a couple of hundred more episodes those two cells would have been visited often enough for the agent to learn the optimal behaviour.

At the point shown in the figure, either the state of individual entities can be saved or the state of the entire experiment can be saved (using Java object serialization mechanism) to be resumed at some later time. The experiment can also be abandoned by quitting the main JAGUAR window. Even in this latter case, however, the logs that the entities have written out will remain, permitting a later analysis by the experimenter.

D.3 Open-source JAGUAR

With the hope of JAGUAR gaining acceptance among researchers interested in Interaction Learning we have turned JAGUAR into an open-source project through the `www.java.net` web site. We believe that this will give JAGUAR the best chance of remaining useful and thriving even after the defense of this thesis. The latest information, including the status of the project, the on-going efforts, the latest news and announcements as well as the planned future work are all available on the project page `http://jaguar.dev.java.net/`.

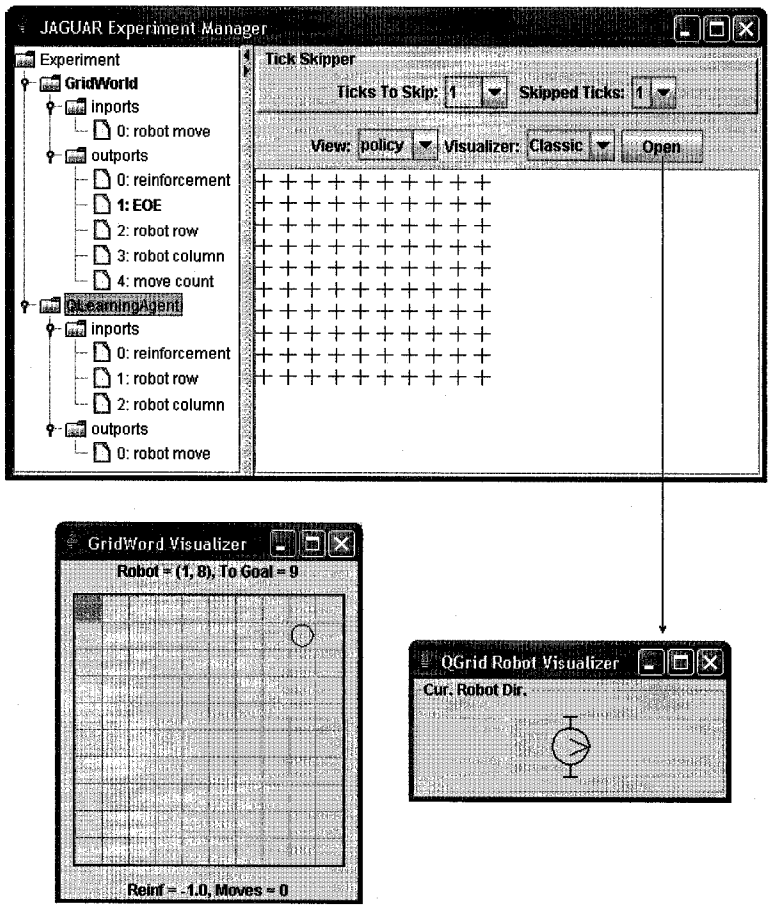

Figure D.2: Both entities are created and connected, so that the experiment is ready to start. The policy view of the agent shows that all four actions for each grid cell (the cross) are equally likely. The visualizers for both entities have also been opened.

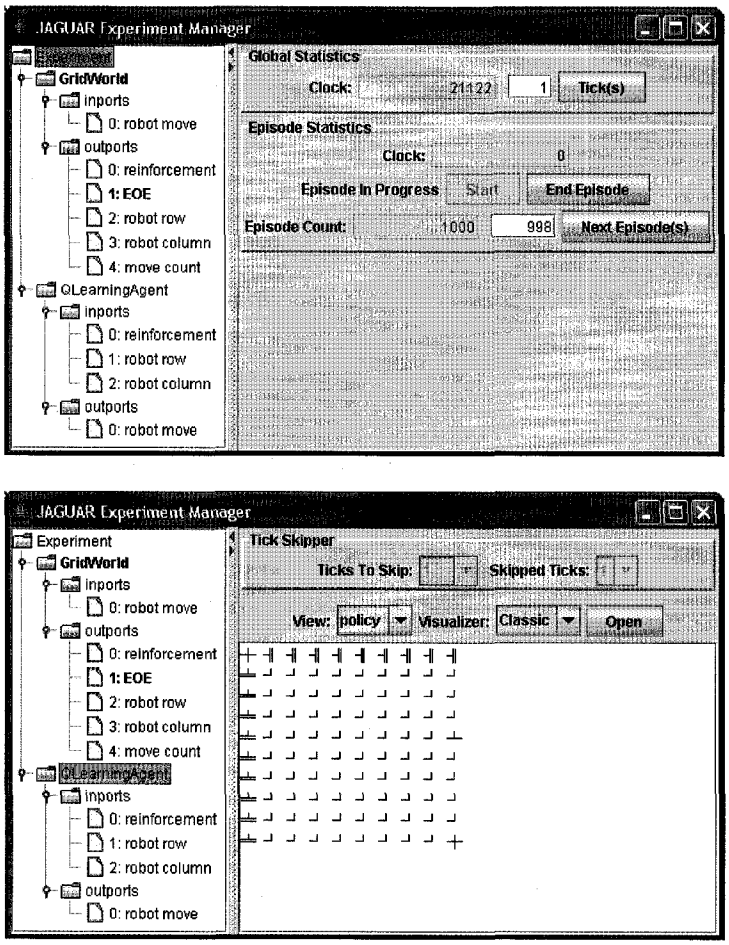

Figure D.3: This figure shows the experiment control panel and the Q-learning agent view after a 1000 episodes had been simulated since the start of the experiment. Note how the policy had nearly converged to the expected solution, and in only two states is the agent unaware of the optimal behaviour.

Bibliography

[1] M. Agache and B. J. Oommen. Generalized pursuit learning schemes: New families of continuous and discretized learning automata. *IEEE Transactions on Systems, Man and Cybernetics: Part B*, 32:738 – 749, 2002.

[2] S. Albers and M. Mitzenmacher. Average case analyses of list update algorithms, with applications to data compression. *Algorithmica*, 21:312 – 329, 1998.

[3] S. Albers and J. Westbrook. Self-organizing data structures. In Amos Fiat and Gerhard Woeginger, editors, *Online Algorithms: The State of the Art*, pages 13 – 51. Springer LNCS 1442, 1998.

[4] J.S. Albus. A theory of cerebellar function. *Mathematical Biosciences*, 10:25 – 61, 1971.

[5] J.S. Albus. *Brain, Behavior, and Robotics.* Byte Books, Peterborough, NH, 1981.

[6] Abdelrahman Amer. Hierarchical list organizing strategies for nonstationary environments. Master's thesis, Carleton University, Ottawa, Ontario, Canada, 2004.

[7] E. J. Anderson, P. Nash, and R. R. Weber. A counterexample to a conjecture on optimal list ordering. *Journal of Applied Probability*, 19(3):730 – 732, 1982.

[8] Marlow Anderson and Todd Feil. Turning lights out with linear algebra. *Mathematics Magazine*, 71(4):300 – 303, 1998.

[9] J.H. Andreae. Approximate dynamic programming for real-time control and neural modeling. *Encyclopedia of Information, Linguistics and Control, A.R. Meetham and R.A. Hudson (eds.)*, pages 261 – 270, 1969.

[10] A.G. Barto and S. Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems: Theory and Applications*, 13:41 – 77, 2003.

[11] Denis V. Batalov. Interaction protocol for interaction learning. *CITO Inaugural Researcher Retreat*, 1998.

[12] Denis V. Batalov. Understanding goals in learning by interaction. *Proceedings of the 2002 IEEE World Congress on Computational Intelligence*, pages 1510 – 1515, 2002.

[13] Denis V. Batalov and B. John Oommen. On playing games without knowing the rules. *Proceedings of the Joint Ninth IFSA World Congress and Twentieth NAFIPS International Conference*, pages 1862 – 1868, 2001.

[14] Denis V. Batalov and B. John Oommen. Turning lights out with dq-learning. *Proceedings of the 24th IASTED International Multi-Conference on Artificial Intelligence and Applications*, February 2006.

[15] R.E. Bellman. *Dynamic Programming.* Princeton University Press, Princeton, 1957.

[16] R.E. Bellman. A markov decision process. *Journal of Mathematical Mechanics*, 6:679 – 684, 1957.

[17] J. L. Bentley and C. C. McGeoch. Amortized analyses of self-organizing sequential search heuristics. *Communications of the ACM*, 28(4):404 – 411, 1985.

[18] D.P. Bertsekas. *Dynamic Programming and Optimal Control.* Athena Scientific, Belmont, MA, 1995.

[19] D.P. Bertsekas and J.N. Tsitsiklis. *Neuro-Dynamic Programming.* Athena Scientific, Belmont, MA, 1996.

[20] J. R. Bitner. Heuristics that dynamically organize data structures. *SIAM Journal on Computing*, 8(1):82–110, 1979.

[21] D.S. Broomhead and D. Lowe. Multivariable functional interpolation and adaptive networks. *Complex Systems*, 2:321 – 355, 1988.

[22] P. Chassaing. Optimality of move-to-front for self-organizing data structures. *Annals of Applied Probability*, 3(4):1219 – 1240, 1993.

[23] F. R. K. Chung, D. J. Hajela, and P. D. Seymour. Self-organizing sequential search and hilbert's inequality. In *Proceedings of the 17th Annual Symposium on the Theory of Computing*, pages 217 – 223, 1985.

[24] R.H. Crites and A.G. Barto. Improving elevator performance using reinforcement learning. *Advances in Neural Information Processing Systems: Proceedings of the 1995 Conference*, pages 1017 – 1023, 1995.

[25] Spears W.M. & Gordon D.F. DeJong K.A. Using genetic algorithms for concept learning. *Machine Learning*, 13:161 – 188, 1993.

[26] J. Dong. Time reversible self-organizing sequential search algorithms. Master's thesis, Carleton University, 1997.

[27] L.P. Kaelbling et al. Reinforcement learning: a survey. *Journal of Artificial Intelligence Research*, 4:237 – 285, 1996.

[28] Robert Holte et al. Searching with abstractions: A unifying framework and new high-performance algorithm. *Proceedings of the Canadian Artificial Intelligence Conference*, pages 263 – 270, 1994.

[29] LiMin Fu. *Neural Networks in Computer Intelligence.* McGraw-Hill, 1994.

[30] J. Grossman, J. Brown, and T. Knight. A lightweight idempotent messaging protocol for faulty networks. In *Proceedings of the 2002 Symposium on Parallel Algorithms and Architectures*, 2002.

[31] Kevin Gurney. *An Introduction to Neural Networks.* UCL Press, University of Sheffield, 1997.

[32] W. J. Hendricks. An account of self-organizing systems. *SIAM Journal on Computing*, 5(4):715 – 723, 1976.

[33] J. H. Hester and D. S. Hirschberg. Self-organizing linear search. *ACM Computing Surveys*, 17(3):295–311, 1985.

[34] Alan Hutchinson. *Algorithmic Learning.* Oxford University Press, Oxford, England, 1994.

[35] F.V. Jensen. *An Introduction to Bayesian networks.* Springer Verlag, New York, 1996.

[36] P. Kanerva. *Sparse Distributed Memory.* MIT Press, Cambridge, MA, 1988.

[37] D. E. Knuth. *The art of computer programming*, volume 3. Addison-Wesley, 1973.

[38] J.L. Kolodner. *Case-Based Reasoning.* Morgan Kaufmann, San Francisco, 1993.

[39] Richard E. Korf. Real-time heuristic search. *Artificial Intelligence*, 42(3):189 – 211, 1990.

[40] V.I. Krinsky. An asymptotically optimal automaton with exponential convergence. *Biofizika*, 9:484 – 87, 1964.

[41] V.Yu. Krylov. One stochastic automaton which is asymptotically optimal in random medium. *Automation and Remote Control*, 24:1114 – 16, 1964.

[42] J.K. Lanctôt and B.J. Oommen. Discretized estimator learning automata. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-22, November/December:1473 – 1483, 1992.

[43] Doug Lenat. Cyc: A large-scale investement in knowledge infrastructure. *Communications of the ACM*, 38(11):33 – 38, 1995.

[44] Oscar Martin-Sanchez and Crisobal Pareja-Flores. Two reflected analyses of lights out. *Mathematics Magazine*, 74(4):295 – 304, 2001.

[45] J. McCabe. On serial files with relocatable records. *Operations Research*, 12:609 – 618, 1965.

[46] Donald Michie. Trial and error. *Science Survey, Part 2, S.A. Barnett and A. McLaren (eds.)*, pages 129 – 145, 1961.

[47] Chambers R. Michie D. Boxes: an experiment in adaptive control. *Machine Intelligence 2, ed. Dale E., Michie D.*, pages 137 – 152, 1968.

[48] M.L. Minsky. Steps towards artificial intelligence. *Proceedings of the Institute of Radio Engineers*, 49:8 – 30, 1961.

[49] Tom M. Mitchell. Version spaces: A candidate elimination approach to rule learning. *Proceedings of the Fifth International Joint Conference on AI*, pages 305 – 310, 1977.

[50] Tom M. Mitchell. *Machine Learning.* McGraw-Hill, New York, 1997.

[51] K. S. Narendra and M. A. L. Thathachar. *Learning Automata: An Introduction.* Prentice Hall, 1989.

[52] K.S. Narendra and R.M. Wheeler. Decentralized learning in finite markov chains. *IEEE Transactions on Automatic Control*, AC31(6):519 – 526, 1986.

[53] Kumpati S. Narendra and Mandayam A.L. Thathachar. *Learning Automata, An Introduction.* Prentice Hall, 1989.

[54] Nils J. Nilsson. Human-level artificial intelligence? be serious! *AI Magazine*, 26(4):68 – 75, 1950.

[55] N.J. Nilsson. *Principles of Artificial Intelligence.* Morgan Kaufmann Publishers, Los Altos, California, 1980.

[56] B. J. Oommen. Absorbing and ergodic discretized two-action learning automata. *IEEE Transactions on Systems, Man, and Cybernetics*, 16(2):282–293, 1986.

[57] B. J. Oommen and M. Agache. Continuous and discretized pursuit learning schemes: Various algorithms and their comparison. *IEEE Transactions on Systems, Man and Cybernetics: Part B*, 31:277 – 287, 2001.

[58] B. J. Oommen and J. R. P. Christensen. Epsilon-optimal discretized reward-penalty learning automata. *IEEE Transactions on Systems, Man and Cybernetics: Part B*, 18:451 – 458, 1988.

[59] B. J. Oommen and J. Dong. Generalized swap-with-parent schemes for self-organizing sequential linear lists. *Proceedings of ISAAC'97, the 1997 International Symposium on Algorithms and Computation*, pages 414 – 423, 1997.

[60] B. J. Oommen, E. R. Hansen, and J. I. Munro. Deterministic optimal and expedient move-to-rear list organizing strategies. *Theoretical Computer Science*, 74:183 – 197, 1990.

[61] B. J. Oommen and D. C. Y. Ma. Deterministic learning automata solutions to the equipartitioning problem. *IEEE Transactions on Computers*, 37(1), 1988.

[62] P.Dayan. The convergence of td(λ) for general λ. *Machine Learning*, 8:341 – 362, 1992.

[63] Roger Penrose. *Shadows of the Mind.* Oxford University Press, 1994.

[64] V.A. Ponomarev. A construction of an automaton which is asymptotically optimal in a stationary random medium. *Biofizika*, 9:104 – 10, 1964.

[65] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81 – 106, 1986.

[66] J.R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, San Mateo, CA, 1993.

[67] N. Reingold, J. Westbrook, and D. D. Sleator. Randomized competitive algorithms for the list update problem. *Algorithmica*, 11(1):15 – 32, 1994.

[68] R. Rivest. On self-organizing sequential search heuristics. *Communications of the ACM*, 19(2):63 – 67, 1976.

[69] H. Robbins. Some aspects of the sequential design of experiments. *Bulletin of American Mathematical Society*, 58(5):527 – 35, 1952.

[70] S. Ross. *Introduction to Stochastic Dynamic Programming.* Academic Press, New York, 1983.

[71] S. M. Ross. *Introduciton to Probability Models.* Academic Press, London, 2 edition, 1980.

[72] R.S.Sutton. Learning to predict by the method of temporal differences. *Machine Learning*, 3:9 – 44, 1988.

[73] L. De Raedt S. Dzeroski and K. Driessens. Relational reinforcement learning. *Machine Learning*, 43(1-2):7 – 52, 2001.

[74] A. L. Samuel. Some studies in machine learning using the game of checkers ii — recent progress. *IBM Journal R & D*, 11(6):601 – 617, 1967.

[75] John R. Searle. Minds, brains, and programs. *The Behavioral and Brain Sciences*, 3:417 – 424, 1980.

[76] I.J. Shapiro and K.S. Narendra. Use of stochastic automata for parameter self-optimization with multi-modal performance criteria. *IEEE Transactions on Systems Science and Cybernetics*, 5:352 – 360, 1969.

[77] Alexander Shen. Mathematical entertainments: Lights out. *Mathematical Intelligencer*, 22(13):20 – 21, 2000.

[78] S.P. Singh and R.S.Sutton. Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22:123 – 158, 1996.

[79] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[80] Robert J. Sternberg and Douglas K. Detterman (eds.). *What is intelligence? Contemporary viewpoints on its nature and definitions.* Ablex, Norwood, NJ, 1986.

[81] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning, An Introduction.* MIT Press, 1998.

[82] A. M. Tenenbaum. Simulations of dynamic sequential search algorithms. *Communications of the ACM*, 21(9):790 – 791, 1978.

[83] Gerald Tesauro. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 5(2):215 – 219, 1994.

[84] M.A.L. Thathachar and B.J. Oommen. Discretized reward-inaction learning automata. *Journal of Cybernetics and Information Sciences*, pages 24 – 29, Spring, 1979.

[85] M.A.L. Thathachar and P.S.Sastry. Estimator algorithms for learning automata. *Proc. Platinum Jubilee Conf. Syst. Signal Processing*, 1986.

[86] M.A.L. Thathachar and P.S.Sastry. *Networks of Learning Automata: Techniques for Online Stochastic Optimization.* Kluwer Academic Publishers, 2004.

[87] M.L. Tsetlin. On the behaviour of finite automata in random media. *Avtomatika i Telemekhanika*, 22:1345 – 54, 1961.

[88] M.L. Tsetlin. *Automaton Theory and Modeling of Biological Systems.* Academic Press, New York, 1973.

[89] Alan M. Turing. Computing machinery and intelligence. *Mind*, 59:433 – 466, 1950.

[90] V.I. Varshavskii and I.P. Vorontsova. On the behaviour of stochastic automata with a variable structure. *Automation and Remote Control*, 24:327 – 33, 1963.

[91] C. Watkins. *Learning from delayed rewards.* PhD thesis, Cambridge University, Kings College, Cambridge, England, 1989.

[92] P.J. Werbos. Approximate dynamic programming for real-time control and neural modeling. *Handbook of Intelligent Control: Neural, Fuzzy and Adaptive Approaches, D.A. White and D.A. Sofge (eds.)*, pages 493 – 525, 1992.