Evolutionary Reinforcement Learning of Spoken Dialogue Strategies

Dave Toney

Doctor of Philosophy

Institute for Communicating and Collaborative Systems
School of Informatics
University of Edinburgh

2007

Abstract

From a system developer's perspective, designing a spoken dialogue system can be a time-consuming and difficult process. A developer may spend a lot of time anticipating how a potential user might interact with the system and then deciding on the most appropriate system response. These decisions are encoded in a dialogue strategy, essentially a mapping between anticipated user inputs and appropriate system outputs.

To reduce the time and effort associated with developing a dialogue strategy, recent work has concentrated on modelling the development of a dialogue strategy as a sequential decision problem. Using this model, reinforcement learning algorithms have been employed to generate dialogue strategies automatically. These algorithms learn strategies by interacting with simulated users. Some progress has been made with this method but a number of important challenges remain. For instance, relatively little success has been achieved with the large state representations that are typical of real-life systems. Another crucial issue is the time and effort associated with the creation of simulated users.

In this thesis, I propose an alternative to existing reinforcement learning methods of dialogue strategy development. More specifically, I explore how XCS, an *evolutionary* reinforcement learning algorithm, can be used to find dialogue strategies that cover large state spaces. Furthermore, I suggest that hand-coded simulated users are sufficient for the learning of useful dialogue strategies. I argue that the use of evolutionary reinforcement learning and hand-coded simulated users is an effective approach to the rapid development of spoken dialogue strategies.

Finally, I substantiate this claim by evaluating a learned strategy with real users. Both the learned strategy and a state-of-the-art hand-coded strategy were integrated into an end-to-end spoken dialogue system. The dialogue system allowed real users to make flight enquiries using a live database for an Edinburgh-based airline. The performance of the learned and hand-coded strategies were compared. The evaluation results show that the learned strategy performs as well as the hand-coded one (81% and 77% task completion respectively) but takes much less time to design (two days instead of two weeks). Moreover, the learned strategy compares favourably with previous user evaluations of learned strategies.

Acknowledgements

I wish to thank my thesis supervisors, Johanna Moore and Oliver Lemon, for their expertise, guidance and perseverance. I also wish to acknowledge the assistance of members of the classifier system community, particularly Tim Kovacs, Pier Luca Lanzi and Stewart Wilson. I am very grateful to my family, particularly my parents, for their encouragement over many years. Finally, I want to express my heartfelt gratitude to my wife, Clare. Without her support – professional, financial and emotional – I could not have undertaken and completed my PhD.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. The length of this thesis, including the bibliography, is approximately 41,000 words.

(*Dave Toney*)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Since the early 1990s, speech and language-processing technologies have been combined to create spoken dialogue systems (SDSs). These interfaces allow users to converse with a computer to retrieve information, conduct transactions, or engage in problem-solving tasks (McTear, 2004). From a developer's perspective, designing a SDS can be a difficult and time-consuming process. Although each SDS is normally designed for a specific purpose (e.g. to provide a weather report), the number of unique conversations that can occur between a user and the system is almost unlimited. Consequently, a system developer may spend a lot of time anticipating how a potential user might interact with the system and then deciding on the most appropriate system response (Zue and Glass, 2000). These decisions are encoded in a *dialogue strategy*, essentially a mapping between anticipated user inputs and appropriate system outputs.

In practice, the strategy is often limited by the capabilities of the available technology (Cavazza, 2001). Nevertheless, the task of encoding anticipated user behaviour and system responses is a complex one. Another problem for the system developer is how to assess whether the chosen dialogue strategy is the optimal one. In other words, does the strategy achieve the best possible compromise between the needs of potential users and the limitations of the available technology? A number of usability criteria have been proposed to help developers evaluate their strategies (Walker et al., 1998; Paek, 2001). However, these criteria do not explicitly suggest the modifications that

will produce optimal strategies. Instead, developers analyse evaluation results and then employ their skill and experience to improve their dialogue strategies. For a particular system, this test-and-refine process may be repeated many times.

How can a developer reduce the time and effort required to design a spoken dialogue strategy? How can s/he have confidence that the chosen dialogue strategy is the best possible one? Put simply, how can a developer *build a good strategy quickly*? One approach to addressing both of these questions is to model the search for a dialogue strategy as a sequential decision problem and then use reinforcement learning techniques to find a solution to the problem (Levin et al., 2000; Pietquin and Dutoit, 2006). This approach typically requires the creation of a simulated user, a model of how typical users might interact with a SDS (Georgila et al., 2005).

Progress has been made within this framework but some important challenges remain. For instance, little success has been achieved with the large state representations that are typical of real-life systems (Henderson et al., 2005). Another crucial issue is the time and effort associated with the creation of simulated users (Schatzmann et al., 2006). Often, simulated users are created by applying supervised learning methods to either: (i) a relevant, pre-existing corpus or (ii) a corpus collected specifically for the intended application. An obvious dilemma is whether the time taken to train a simulated user might detract from the goal of developing dialogue strategies quickly.

1.2 Aims

This thesis has three main aims. Firstly, I wish to investigate an alternative to existing reinforcement learning techniques for developing dialogue strategies. More specifically, I will explore how XCS, an evolutionary reinforcement learning algorithm, can be used to generate strategies. The XCS algorithm possesses a number of useful qualities that make it an appealing candidate for the learning of dialogue strategies. For instance, the algorithm's rule-based representation is a good match for how many dialogue strategies are represented, particularly information-seeking systems. Additionally, a number of theoretical and empirical studies demonstrate that XCS can generate rule sets that are optimal and maximally general with respect to sequential decision tasks (Kovacs, 2002; Lanzi, 2002). Most importantly, the algorithm can generate rule sets covering very large state spaces (Wilson, 1995).

Secondly, I will determine whether useful dialogue strategies can be developed without the use of a training corpus. Naturally, a classifier trained on a corpus of relevant human-computer, or even human-human dialogues, has more validity than a simulated user which is hand-coded by a system developer. However, if we are to build novel spoken dialogue applications then new corpora must be collected. This takes a lot of time (Walker et al., 2001). Hand-coded simulated users require much less time to create, and even refine (Chung, 2004). Therefore, an important question that needs to be answered is whether hand-coded simulated users are sufficient for the generation of useful dialogue strategies.

Finally, the performance of a spoken dialogue strategy can only be properly assessed by: (i) integrating it into a complete dialogue system; and (ii) having the system's performance evaluated by human users. Therefore, a third aim of this thesis is to empirically evaluate a dialogue strategy generated by XCS in conjunction with a hand-coded simulated user. The strategy will be integrated into an end-to-end spoken dialogue system. The system will allow human users to complete a realistic task.

1.3 Outline

The remainder of the thesis is structured as follows: *Chapter 2* gives an overview of spoken dialogue systems and the issues surrounding the development of dialogue strategies. This chapter also surveys the current approaches to generating dialogue strategies automatically; *Chapter 3* outlines a framework for developing dialogue strategies using evolutionary learning and hand-coded simulated users; *Chapter 4* presents a series of experimental results demonstrating the feasibility of the proposed framework; *Chapter 5* describes the application of the framework to a more realistic problem, requiring a very large state-action space; *Chapter 6* reports on an evaluation by real users of the framework as part of a complete spoken dialogue system; *Chapter 7* lists a summary of the contributions made by the thesis and provides suggestions for future research.

Chapter 2

Background

2.1 Overview

This chapter provides a brief introduction to spoken dialogue systems (SDSs). I list some of the spoken dialogue applications that have been developed in the last few years and also describe the system architecture of a typical SDS. I then focus on one particular component of a SDS: the dialogue strategy. The dialogue strategy defines how a system should interpret a user's intentions and generate appropriate responses. Developing an effective strategy often demands a great deal of time and expertise on the part of a system developer. I outline what is required of dialogue strategies, the main ways in which they are implemented, and how they are evaluated.

Recent work has focused on generating dialogue strategies automatically and this will be reviewed here. The basic premise of this work is that a dialogue can be modelled as a Markov Decision Process. Reinforcement learning techniques can then be applied to this model in order to generate dialogue strategies. I describe in detail a commonly used algorithm: Q-learning. This is followed by a case study to help illustrate the ideas discussed so far. An important component of developing learned strategies is the use of dialogue simulations. A summary of the most recent work in dialogue simulation is presented. Finally, I review the main challenges associated with learning dialogue strategies and discuss the progress that has been made towards addressing these issues.

2.2 Spoken dialogue systems

A spoken dialogue system (SDS)[1] is a natural language interface designed to make use of spoken language technology for both language understanding and generation and so to effect a conversation between a user and computer. Such a system usually provides access to a computer-based application such as a database or an expert system. The development of SDSs has been made possible by improvements in speech technology in recent years. However, there is still a long way to go before a user can converse in a completely natural manner. Despite the rapid progress achieved, dialogue system developers must often make compromises between naturalness and recognition accuracy. One of the subtleties of dialogue system design lies in giving users a feeling of naturalness and freedom of response in spite of the underlying system constraints. Nevertheless, usable and effective applications are being developed at an increasing rate.

2.2.1 Applications

A wide variety of spoken dialogue systems have been implemented; they include: (i) information retrieval and transactional systems; (ii) intelligent assistants; and (iii) tutorial systems.

Information retrieval and transactional systems are arguably the most prevalent forms of spoken dialogue systems in use today. Interaction is almost exclusively telephony-based. Existing applications allow users to retrieve tourist, weather and telephone directory information (Litman et al., 2000; Zue et al., 2000), make travel, restaurant, cinema and theatre reservations (Seneff and Polifroni, 1996; Shriver, 2001) and conduct financial transactions (McInnes et al., 1999). At the simplest level, these applications are a speech-based version of form-filling. That is, a system elicits some parameters from the user and then executes a database query. However, the results of such a query may be too large to report sequentially in spoken form. Therefore, increasingly sophisticated strategies have been developed for presenting users with the appropriate information (Demberg and Moore, 2006). Figure 2.1 presents an example of an interaction between a user and a train information system (Aust et al., 1995).

[1] Spoken dialogue systems are also known as conversational interfaces or spoken language systems.

System	Good morning. This is the automatic timetable information service. How can I help you?
User	Good morning. I need a connection from Hamburg to Munich.
System	When would you like to go from Hamburg to Munich?
User	Next Friday, er, at eight o'clock.
System	So you want to travel on Sunday the 25th of September at 8 a.m.?
User	No, on Friday.
System	So you want to travel on Friday the 23rd of September at 8 a.m.?
User	Exactly.
System	There is the following connection. With Intercity Express 793, departure from Hamburg at 8:02, arrival in Munich at 14:05. Would you like me to repeat the connection?
User	No.
System	I wish you a pleasant journey. Thank you for calling. Goodbye.

Figure 2.1: Example of a train information dialogue (Aust et al., 1995).

Intelligent assistants support human users in complex decision-making tasks. By having humans and machines work together, rather than in isolation, it is suggested that better solutions to some problems can be found (Allen and Ferguson, 2002). For instance, the TRAINS and TRIPS projects at the University of Rochester are long-term efforts to develop an intelligent natural language planning system (Allen et al., 1996, 2000). In the TRAINS system, a scheduling assistant cooperates with a human supervisor in the management of railroad freight. In the TRIPS system, an assistant collaborates with a human manager to construct an evacuation plan for a fictitious island that is about to encounter a hurricane.

Tutorial systems allow users to converse with a knowledge base in order to acquire new skills or learn new concepts. These systems make inferences about a user's understanding of topics in order to dynamically alter the content of one-to-one tutor style of instruction. The development of tutorial systems is motivated by the observation that one-to-one tutoring often yields significantly higher gains than classroom instruction (Bloom, 1984). Tutorial systems attempt to replicate many of the advantages of human tutors in a cost-effective manner. Presently, most systems are text-based, but work has begun on developing speech-based tutors (Litman, 2002; Pon-Barry et al., 2004).

In all of these applications, the interaction between system and user consists of a cycle of spoken utterances. In many conversational systems, the user is able to interrupt the system while it is speaking. In more dynamic, event-driven environments, it is possible for the system to deliberately interrupt the user (Lemon et al., 2001). The conversational cycle continues until the user's goal has been achieved or until the user decides to terminate or suspend the conversation. In a simple information retrieval task, an interaction may be completed in just a few system-user exchanges. In sophisticated planning or tutorial systems, a conversation may last for several hours. With some systems, speech is integrated with other forms of input (e.g. pointing gestures on a touch-sensitive screen). These more advanced interfaces, often referred to as multimodal systems, are beyond the scope of this thesis (but see Oviatt and Cohen (2000); Brøndsted et al. (2002) for reviews).

2.2.2 System architecture

From a user's perspective, the conversational cycle is conducted with a single computer system. In reality, implementing a SDS requires the integration of several speech- and language-processing components (Figure 2.2). Normally, a SDS will include a speech recogniser, a speech synthesiser and possibly natural language understanding and generation components. An excellent introduction to these technologies can be found in Jurafsky and Martin (2000).

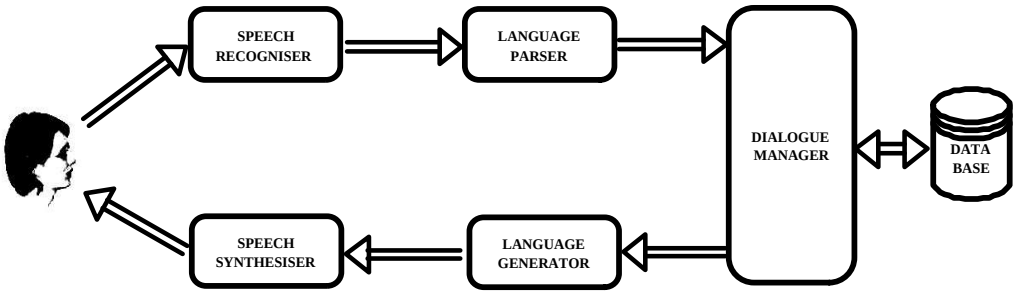

Figure 2.2: Conceptual outline of a spoken dialogue system.

The basic stages of processing in a single conversational cycle are: (i) the user's utterance in the form of a speech signal is sampled and processed by the speech recogniser; (ii) the recogniser outputs a list of potential sentences (hypotheses),

with associated confidence levels, to the language parser; (iii) the parser examines these hypotheses and decides on a meaning that can be usefully interpreted by the dialogue manager; (iv) the dialogue manager analyses the parser's output in the context of the dialogue as a whole; it then decides on the most appropriate response, possibly retrieving information from the database in the process; (v) this response is converted into a complete sentence by the language generator; (vi) finally, the speech synthesiser translates the text from the language generator into spoken form.

2.2.3 Spoken dialogue strategies

In the previous section, I characterised the dialogue manager as a module that analyses the output from the language parser and then decides on an appropriate response for output to the language generator. In practice, this module is executing a *dialogue strategy*, created by the system developer. This strategy defines how the system should interpret the user's intentions and how it should generate its responses. Dialogue strategies can vary widely in terms of how they are represented and in their level of sophistication. This thesis addresses the difficulties associated with developing a dialogue strategy and how these difficulties can be alleviated.

2.3 Developing spoken dialogue strategies

This section outlines the way in which spoken dialogue systems are developed and the challenges that are faced by system developers. Typically, a system's development begins with an initial specification from a client. In the context of SDSs, a client is someone who wishes to provide human users with access to a speech-based interface to perform a particular task. For example, an airline company may want to create a SDS for ticket reservations, allowing customers to make reservations over the telephone, 24 hours a day without the need to speak with a human operator. The client's knowledge of the intended task, domain and customers is often sufficient for a developer to begin implementing a dialogue strategy. Frequently, a developer draws upon a number of other resources to influence the design of this initial strategy (Figure 2.3). For example, there may be previously recorded examples of human-computer or human-human dialogues that are relevant to the task at hand.

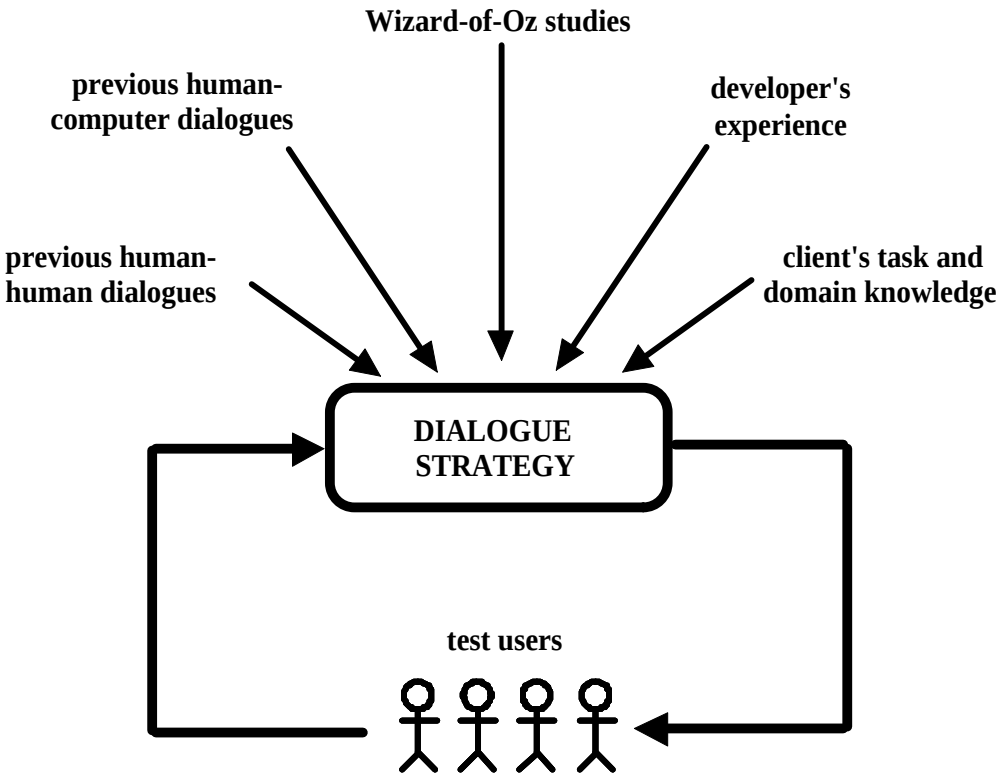

Figure 2.3: Possible resources for the design of a dialogue strategy.

If these are not available, the developer might devise a simple prototype to facilitate the collection of such data (often referred to as a *Wizard-of-Oz* study, see Fraser and Gilbert, 1991; Klemmer et al., 2000). In addition, the developer can make use of his/her own experience of end users and of the available technology. When the first pass at a dialogue strategy has been implemented, it is usually evaluated by a group of test users and then amended in light of the evaluation results. This test-and-refine procedure may be repeated many times before the system is deployed for widespread use. The deployed system may itself be periodically reviewed and amended if necessary. This development lifecycle will now be described in more detail with respect to three important questions: (i) what makes for a good dialogue strategy; (ii) how is a dialogue strategy implemented; and (iii) how does a developer evaluate the chosen strategy?

2.3.1 Strategy requirements

Clearly, the primary goal of any dialogue strategy is to satisfy the functional requirements of the client. That is, the developed strategy should accurately model

the intended task and facilitate users in the completion of this task. However, a conversational system should also address a number of more generic design requirements. As SDSs have been developed in recent years a variety of important design requirements have been identified. Some of the most commonly addressed issues are highlighted below. A fuller treatment of these issues, and others, can be found in Zue and Glass (2000) and van Kuppevelt et al. (2000).

A fundamental problem for developers of SDSs is the constraints associated with today's speech recognition and language-processing technologies. This means that users cannot yet interact with SDSs using the same range of vocabulary and contextual knowledge that is typical of human-human conversations. To be successful, a dialogue strategy must strike a balance between the system's technological capabilities and a user's desire for a natural conversation. Therefore, it should be made clear to a user what the capabilities of the system actually are (Yankelovich, 1996). This is particularly important for users who are unfamiliar with speech interfaces. If their only reference point is human-human conversations they may become quickly frustrated by the system's limited abilities. How can this be avoided? One way is to carefully choose system prompts that encourage users to respond in a way that the system can understand (e.g."Do you want to transfer some money or order a new chequebook?"). This system-directed policy is a common dialogue strategy and can work well in many situations (Hone and Baber, 2001). However, if used repeatedly, it can quickly become tiresome for the user.

An alternative is to ask more open-ended questions and allow users to take the initiative in deciding how the task is accomplished (e.g. "Which service would you like?"). There are two risks associated with this tactic. Firstly, with more options available to the user at each point in the dialogue, the level of misrecognition on the part of the system may be higher. This is likely to frustrate the user. Secondly, the user may lose track of what has been accomplished in a dialogue and become unable to find a way of reaching the intended goal. A natural compromise between tightly controlled and completely open systems is to allow *mixed-initiative*. In other words, a system can provide the user with a pre-defined path but also allow the user the flexibility to deviate from this path and reach the intended goal by an alternate route (Ramakrishnan et al., 2002). This is a useful way to accommodate both novice users, who may need guidance, and 'expert' users who use the system frequently and so know the shortcuts to achieving their goal.

Another important issue for SDSs is the notion of *grounding*. Conversational participants are said to "share a common ground" if they have a mutual understanding of what is occurring in the dialogue (Clark, 1996). To operate effectively, it is essential for a SDS to maintain an understanding of the user's beliefs and intentions. When this shared understanding seems out of alignment, the system should take measures to recover from the misunderstanding. A common strategy is to confirm the user's intentions (e.g. "So you want travel from Paris to New York on 10th March. Is that correct?"). Detecting and repairing miscommunications between the system and user is a critical aspect of developing a dialogue strategy (McRoy, 1998). Where possible, a system should make use of as much task, domain and world knowledge as possible. For example, if a system believes that a user wants to fly from London to London, a sensible strategy would immediately trigger a clarification question from the system.

Lastly, a SDS is likely to be used by a highly variable population of human users. In other words, users will differ in their experiences, abilities, preferences and characteristics. A SDS is often more effective if it can infer some of this information from the user and exploit it to personalize its interaction with the user. This process is often referred to as *user modelling* (van Zanten, 1999). It is particularly important in tutoring systems; a fundamental goal of a tutoring system is to adapt its responses in line with the understanding demonstrated by the student (Shute, 1995). However, all conversational systems can be enhanced by adapting their behaviour in accordance with a model of the user (Zadrozny et al., 2000).

In summary, a spoken dialogue system should demonstrate an understanding of the user's intentions, taking and relinquishing control of the conversation where appropriate. It should also avoid, or attempt to repair, any miscommunication between it and the user. It should also try to tailor its behaviour to the user's individual needs and characteristics. In other words, it should behave, as much as possible, like a human agent. Addressing these issues when devising a dialogue strategy can greatly enhance the system's performance and acceptance by human users. However, incorporating these requirements into a system design contributes to the problem of anticipating what a user might say and then selecting the appropriate system responses.

2.3.2 Strategy implementation

This section examines how dialogue strategies are actually implemented in a working system. The strategies of early commercial systems were encoded using high-level programming languages such as C/C++. A strategy's control logic could be represented by constructs such as *if-then* statements. The input and output functions of the system were realised through external library calls to the other software components (e.g. a speech recogniser). However, strategies that were represented in this way were often low in readability, flexibility and portability. At the same time, a number of other approaches to representing conversational strategies emerged. These include: (i) finite-state systems; (ii) collaborative planning systems and (iii) information update systems. Each approach is based on a different theoretical view of what constitutes a dialogue and this is reflected in the differing notations for implementing dialogue strategies. These three approaches are described briefly below.

Finite-state systems are characterised by a state transition network that defines the legal paths through a dialogue. Each node in the network represents a *dialogue state* Associated with each dialogue state is a system prompt and a list of permissible user responses. The determination of the next state is a function of both the current state and the user's response. Using this model, a conversation between the system and a user is managed by the traversal of the finite-state network that was previously defined by the system developer. A task reaches completion when the final state is reached. Dialogue strategies are often implemented using a visual language that explicitly represents the finite state network (e.g. Sutton et al., 1998). More recently, a markup language, VoiceXML, has been devised to allow developers to encode strategies. This language makes it easy for designers to provide users with a speech-based interface to web-based information sources (Lucas, 2000; Bennett et al., 2002).

Using the finite-state model, dialogue strategies can be rapidly prototyped, tested and debugged. A visual representation of the finite state network makes it fairly straightforward for a system developer to encode the logic of a dialogue strategy. However, this development methodology is limited by the fact that every change in the conversation must be represented by a transition between two nodes in the network. Consequently, as the task requirements grow, the size and complexity of such a design can rapidly become unwieldy (Aust and Oerder, 1995).

This problem can be mitigated to some degree by the use of self-contained modules that achieve specific subtasks (e.g. to capture date information). In this way, a complete spoken dialogue application can be constructed as a network of subdialogues (Heeman et al., 1998; Ehrlich, 1999). Notwithstanding this, the finite-state model is useful for small, well-defined tasks, and in situations where speech recognition performance may be relatively low (e.g. over a telephone line). Because of this, many of the commercial systems in use today have been developed using the finite-state model.

Collaborative planning systems are based on the idea that conversational participants coordinate their actions towards achieving some shared goal (Grosz and Sidner, 1986). Therefore, a collaborative planning system attempts to discover what the user's goal is and to assist the user with the development of a plan to realise this goal. It should be noted that these systems often depend on a higher level of speech recognition accuracy (e.g. microphone-based) where the range of vocabulary available to the user is often much larger. Also, a larger amount of task and domain knowledge is often incorporated into the system. Consequently, the interaction between human and user is potentially much more sophisticated compared with finite-state systems (Allen et al., 2000). On the other hand, dialogue strategies are correspondingly much more time-consuming to develop. The size of the strategy can also make debugging very difficult.

Information update systems attempt to combine the simplicity of finite-state systems and the sophistication of collaborative planning systems (Larsson and Traum, 2000). These systems model the effect of user utterances on the system's responses as a series of state-action rules; they typically take the form of "*if the dialogue is in state X then take action Y*". The representation of state *X* may range from a simple state variable to a complex feature structure. Using this representation it is easy to increase incrementally the sophistication of the dialogue strategy. However, it can be difficult to know whether all possible situations have been accounted for. In addition, as the number of rules grows, an information update strategy can become increasingly difficult to debug (Bos et al., 2003).

The three approaches outlined above do not represent an exhaustive list of the techniques available for encoding dialogue strategies. It does however, give some idea of the range of strategy sophistication that is possible; each of these approaches is appropriate for different tasks. Many commercial systems are implemented using finite-state languages. Collaborative planning methods are frequently used in

developing intelligent assistants and tutorial systems. The information update method has been used in a wide range of systems. Despite the success of these approaches, they do not address the issue of how to develop the best possible dialogue strategy.

2.3.3 Strategy evaluation

An essential part of developing a software system is to evaluate its performance with test users before deployment in the real world. A wide variety of evaluation criteria and metrics for SDSs have been proposed over the years (e.g. Walker et al., 1998; Paek, 2001). Evaluation metrics can be categorised into two types: objective and subjective. Objective metrics can be calculated automatically from logs of test dialogues. Measurements that are commonly taken include the length of time taken for each dialogue, the number of system and user utterances, the number of times a user asked for help, etc. By contrast, subjective metrics record users' impressions about the quality of the system's performance. This information is usually gathered using a questionnaire after users have interacted with the system. Users are asked to rate the system's performance with respect to a range of criteria, such as intelligibility, ease of use, etc.

As a system is developed, it can be iteratively evaluated by test users and refined by the designer in the light of the evaluation results. However, this can be a very time-consuming and expensive process. In practice, what often happens is that a developer asks some colleagues to experiment with the early prototypes of a system. As system development progresses, the test regime becomes more formal, using more naïve test users. An alternative approach to reducing the cost associated with repeated testing is to create a simulated user. This 'user' is a model of how human users behave (Eckert et al., 1998; López-Cózar et al., 2002). This model can be created using previous examples of dialogues as well as the experience and intuition of the designer and client.

It seems obvious that testing a system with real users is likely to be more accurate. On the other hand, it is often much cheaper to develop a simulated user model, which can be used by the system developer as many times as is required. A satisfactory compromise might be to perform initial tests with a simulated user and then to recruit real users in the final stages of evaluation. The simulated user model has also become

an important ingredient in the development of dialogue strategies based on machine learning techniques, as we shall see in the next section.

2.4 Learning spoken dialogue strategies

Developing a spoken dialogue system is a complex and time-consuming task. Consequently, attempts have been made to automate parts of the development process; the use of simulated user models for system evaluation is one example of this. More recently, researchers have focused on automating the design and implementation stages of system development. The idea is for developers to specify *what* behaviour is required of the system rather than *how* it is to be achieved. That is, the dialogue strategy is automatically generated from the developer's specification. One approach to accomplishing this automated development process is to use reinforcement learning (RL) techniques. To generate a dialogue strategy using RL, we first need to model a human-computer dialogue as a sequential decision task. In particular, by formalising a dialogue as a Markov Decision Process, RL can be used to learn dialogue strategies.

2.4.1 Sequential decision tasks and Markov Decision Processes

Sequential decision tasks are characterised by an agent which interacts with its environment. In the simplest case, the agent interacts with the environment in discrete time steps (see Figure 2.4). At each step, the agent receives some representation of the state of the environment s. Using this information, the agent selects an action a. The effect of this action on the environment is fed back to the agent in two ways. First, the agent will receive a numerical reward r. This reward is used to reinforce – or possibly discourage – the agent's selection of that action the next time it is in that state. Second, the agent will receive a new representation of the environment's state, s', which becomes the new value for s in the next time step (i.e. $s \leftarrow s'$). This process is repeated many times. The purpose of this iterative process is for the agent to learn how to maximise the amount of reward that it receives from its environment. Practical applications of this model include robot navigation (Crook and Hayes, 2003b), board game playing (Tesauro, 1995) and channel allocation for mobile (cellular) telephone networks (Singh and Bertsekas, 1997).

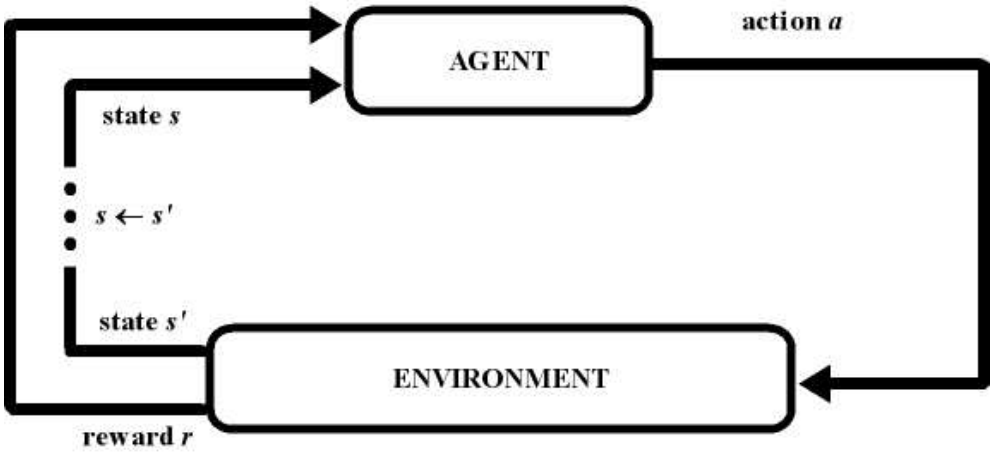

Figure 2.4: The agent-environment interaction in a sequential decision task.

In a sequential decision task, an *episode* is the sequence of steps taken by the agent to reach a pre-defined terminal state from its initial state. In a board game, for example, an episode is a single game. Remembering that the agent receives a reward r at each time step, the goal of the agent is learn how to maximise the total reward received in a complete episode. The strategy developed by the learning agent for achieving this maximal cumulative reward is known as a *policy*. To learn a good policy in a sequential decision task, actions need to be chosen in order to accumulate maximum reward over the entire episode. That is, actions should be chosen for their reward *over the long term*, rather than simply for their immediate reward r. For example, in a chess game, many low reward moves (e.g. sacrificing pieces) may eventually give rise to a very high reward (e.g. checkmate). A good policy is one where actions are chosen for their long term reward.

An important form of sequential decision task is the *Markov Decision Process* (MDP). MDPs are based on the notion that the current state of the environment contains all the relevant information needed to decide on which action to take next. A state representation that fulfills this requirement is said to be *Markov*, or to have the *Markov property*, or to satisfy the *Markov assumption* (Sutton and Barto, 1998, p. 61). For example, in a board game, such as chess or backgammon, the only information an agent actually needs to decide on its next move is the current board position; it does not matter how that board position came about. Put another way, the history of all previous moves are summarised by the current position of the pieces on the board.

This idea can be applied to the concept of *dialogue state*. In many spoken dialogue systems, the dialogue state includes a compact representation of the dialogue's history (see section 2.3.2). Although this state representation may contain information about previous dialogue states, it does not violate the Markov assumption since all the relevant information is summarised by the current dialogue state.

Formally, a finite MDP is defined by a state set $S = \{s_1, ..., s_m\}$, an action set $A = \{a_1, ..., a_n\}$, a transition function $T(s, a, s')$ and a reward function $R(s, a)$. With m possible states and n possible actions, the transition function represents the probability of making a transition to state s' when taking action a in state s, i.e. $P(s'|s, a)$. The reward function represents the immediate reward r associated with taking action a in state s, i.e. $R(s, a) = r$. With these elements defined, we can compute the optimal mapping between states and actions (known as the *optimal policy*[2] in the RL literature). One way to do this is to first compute the *optimal action-value function* Q^*:

$$Q^*(s, a) = E(r|s, a) \quad \forall\, s \in S,\ a \in A. \tag{2.1}$$

For any state s, this function gives the expected total reward if action a is chosen and the interaction proceeds to the final state in an optimal way. This function is essentially a lookup table, indexed by state-action pairs, defining the *value* (long term reward) of selecting individual actions. With this function computed, it is straightforward to induce the optimal policy; we simply look at the actions available in each state and select the action with the highest Q-value. In other words, the optimal policy, π^*, enumerates the optimal action for each state s:

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\, Q^*(s, a) \quad \forall\, s \in S,\ a \in A. \tag{2.2}$$

2.4.2 Reinforcement learning

A variety of algorithms exist for computing the optimal action-value function. These iterative algorithms begin with an initial value function Q_0, where all the state-action values are initialised to some arbitrary value (e.g. 0). In a single iteration, the transition and reward functions are used to update the estimated Q-value of some

[2] For a given task there may be more than one optimal policy. However, we only need to find one of these policies since they are all equally optimal. In the RL literature, it is common simply to refer to *the* optimal policy.

(possibly all) state-action pairs. This new value function Q_1 is a slightly more accurate estimate of Q^*. The update step is repeated many times, with each successive update $(Q_2, Q_3, \ldots, Q_t)$ producing an improved estimate of Q^*.

If the transition function is fully specified, *dynamic programming* can be used to estimate Q^*. This algorithm works by updating the Q-value for every possible state-action pair in a single iteration using the transition and reward functions. Since (i) the transition function defines the probability of moving from any state s to another state s', i.e. $P(s'|s,a)$ and (ii) the reward function $R(s,a)$ defines the reward for choosing the action that creates this transition, the Q-value of each state-action pair can be updated recursively by:

$$Q_t(s,a) \leftarrow R(s,a) + \sum_{s'} P(s'|s,a) \max_{a'} Q_t(s',a') \tag{2.3}$$

Dynamic programming cannot be used when a complete model of the transition function is not available. For example, RL has been applied to the development of backgammon strategies (Tesauro, 1995). There are approximately 10^{20} possible board positions in a backgammon game. This means that the transition matrix enumerating the probability of each possible transition contains 10^{40} elements. Calculating the value of every element in this matrix is computationally infeasible. An alternative approach is to update the estimate of Q^* using *temporal difference (TD) learning*. Instead of considering all possible transitions, TD methods require only sample sequences of state-action transitions.

In backgammon, for example, a sample sequence would be the moves of a complete game. In a dialogue system, a sample sequence would be the system and user utterances of a single dialogue. If we can generate a sufficiently large number of these sequences then the estimate of Q^* can be updated in a manner similar to that used in dynamic programming. The essential difference in the two algorithms is the update rule. In dynamic programming, all possible state transitions are considered. In TD learning, only the sampled transitions contribute to the improved estimate of Q^*:

$$Q_t(s,a) \leftarrow Q_t(s,a) + \alpha[R(s,a) + Q_t(s',a') - Q_t(s,a)] \tag{2.4}$$

TD learning is an example of *model-free* reinforcement learning. By contrast, dynamic programming is a *model-based* technique since it requires a complete model of the environment's behaviour as described by the transition function.

In both forms of RL, the resulting value function is guaranteed to converge on the optimal one if the update rule is repeated an infinite number of times (i.e. $Q_\infty = Q^*$). More practically, a learning algorithm is halted when Q_t is believed to be sufficiently close to Q^* (i.e. $Q_t \approx Q^*$). A variety of criteria have been proposed for deciding when a satisfactory approximation for Q^* has been reached (Kaelbling et al., 1996; Mahadevan, 1996). One method is to halt the algorithm when the absolute difference between successive Q-values for each state-action pair is less than some threshold θ:

$$|Q_t(s,a) - Q_{t-1}(s,a)| < \theta \quad \forall\, s \in S,\ a \in A. \tag{2.5}$$

2.4.3 Q-learning

This section examines the mechanism for estimating Q^* using a well known TD algorithm: Q-learning (Watkins and Dayan, 1992). As shown in Figure 2.5, the algorithm begins with the initialisation of all Q values to some arbitrary value, (e.g. 0). The algorithm terminates when the value function Q is considered to be a satisfactory approximation to the optimal value function Q^*. The value for Q is updated in line 8 according to the states sampled by the agent. In terms of a game-playing task, each iteration of the outside loop represents a single game while each iteration of the inner loop represents a single move. The agent begins each game in some initial state S_I (line 3); the game ends when the agent's action leads to a terminal state S_T (line 10).

Each move of the game is in fact decomposed into five steps (lines 5-9): (i) selection of the move; (ii) execution of the move; (iii) a response from the environment giving the reward associated with the move and the state in which this move will result; (iv) an update of the Q value for the state-action pair that has just been visited; (v) an update of the agent's current state using the feedback received from the environment.

Three lines of the algorithm require further explanation. Firstly, how is action a selected from the set of all available actions (line 5)? A common strategy is to select the action with the highest Q-value most of the time and occasionally (e.g. 10% of the time) select randomly one of the sub-optimal actions. If the agent is not allowed to explore the potential value of sub-optimal actions, the Q function is less likely to converge towards Q*. This strategy is known as an ε-greedy policy, meaning that most of the time an optimal action is chosen, but with probability ε, a random action is selected. Typically, the value of ε is decayed as learning progresses.

```
1   Q(s,a) ← 0  for all s in S, a in A
2   repeat
3      s ← s_I
4      repeat
5         choose an action a from A
6         execute a
7         receive r and s' from the environment
8         Q(s,a) ← Q(s,a) + α[r + γ max_a' Q(s',a') - Q(s,a)]
9         s ← s'
10     until s = s_T
11  until Q ≈ Q*
```

Figure 2.5: Finding an approximation for Q^* using Q-learning, a TD algorithm.

Secondly, in line 7, the value of r is based on the reward function previously defined by a system developer, but how is s' calculated? Although a complete transition model is not required, the agent must have available to it some model of what transitions are *permissible*. For example, it is relatively straightforward for a system developer to encode the rules of backgammon, thus allowing an agent to calculate which new position will result from any chosen action.

Thirdly, how does the update rule in line 8 improve the estimate of Q^*? It can be shown that the closer an action occurs to the end of a task the more its Q value will reflect the probable success of taking that action. For example, we have a better idea of a game's outcome in the later stages of play than we do at the beginning. The general idea of TD algorithms is to pass on the knowledge available in the later states of a task to the earlier states. In other words, the Q values of later states should influence the Q values of earlier states. This process is achieved using the update rule. When the agent is in state s and takes action a the environment gives the state s' that the agent will visit in the next iteration. Of all the actions available in the subsequent state s', the expression $max_{a'}Q(s',a')$ gives the Q value of the optimal action a'. The value of *Q(s,a)* is updated using a combination of the immediate reward it has just received and the value for $Q(s',a')$. It has been shown that this strategy is effective even though the action chosen in the subsequent state s' may not in fact be a' (Watkins, 1989).

The *discount factor* γ, and the *learning rate* [3] α, are positive fractions that control the amount that $Q(s,a)$ moves towards $Q(s',a')$. The discount factor controls how much weight should be attached to future rewards while the learning rate defines how much of the total reward (immediate and future) is incorporated into the current Q-value. Typically, the values for γ and α are decayed as learning progresses.

The effect of the Q-learning algorithm can be illustrated by the simple example given in Table 2.1. The task consists of three states and two actions. Hypothetical values are given for the Q function before and after learning has taken place. The optimal action for each state is the one with the highest Q value after learning. Therefore, the optimal policy is given by: $\pi^*(s_1) = a_2,\ \pi^*(s_2) = a_1,\ \pi^*(s_3) = a_2$. There are a number of issues that have not been discussed in this brief description of reinforcement learning, including the choice of learning parameters and the problems associated with large state spaces. A detailed account of RL algorithms such as dynamic programming, TD learning and others can be found in Sutton and Barto (1998).

state s	action a	$Q(s,a)$ before learning	$Q(s,a)$ after learning
s_1	a_1	0	2.0
	a_2	0	5.6
s_2	a_1	0	3.2
	a_2	0	1.0
s_3	a_1	0	8.4
	a_2	0	9.1

Table 2.1: Action-value function for a simple RL task before and after learning.

2.4.4 Dialogue as a Markov Decision Process

In the introduction, I characterised a dialogue strategy as a mapping between anticipated user inputs and appropriate system outputs. However, a system may need to respond to the same user input in a variety of ways, depending on the context in which it occurs. For example, if a user asks the system for help, the system's response will depend very much on the current state of the dialogue (Gorrell et al., 2002). Therefore, it is more accurate to define a dialogue strategy as a mapping between the set of

[3] The learning rate is also known as the *step-size parameter* (e.g. in Sutton and Barto, 1998).

possible dialogue states *S* (which may include a record of user input) and the set of system actions *A*.

Using this definition, we can translate the goal of automatically generating a dialogue strategy into the search for a mapping between *S* and *A*. But we want to do more than that — we want to find the *optimal* dialogue strategy. In other words, we want to find the optimal mapping between *S* and *A*. If this mapping can be found, a dialogue system will know how best to respond a user's utterances at any point in a dialogue.

This mapping can be found by modelling dialogue as a Markov Decision Process. In terms of the MDP formalism: (i) the *state set* is a list of all possible dialogue states; (ii) the *action set* is the list of possible system acts; (iii) the *transition function* can be estimated from a corpus of previous dialogues or modelled by a simulated user; (iv) the *reward function* is based on evaluation metrics. It is important to remember that an optimal mapping between *S* and *A* is optimal with respect to the reward function. Whether such a mapping produces the best possible dialogue strategy remains an open question.

2.4.5 Case study

As an illustration, I now describe how the MDP approach to dialogue modelling has been used to improve the strategy of a tourist information system: NJFun (Litman et al., 2000; Singh et al., 2002). In this system, the authors took advantage of situations where the appropriate system action was obvious. For example, it seems clear that the first system action in any interaction should be a greeting prompt. The authors hard-coded such system behaviours. This meant that learning need only be applied to more contentious design decisions. In particular, they focused on two important design choices: (i) the type of initiative to use (system, user or mixed) when asking or re-asking the user for a piece of information, and (ii) whether to confirm a value once it was obtained. The full state representation for this dialogue system consisted of 14 variables. However, not all of these variables needed to be considered for the purposes of learning; the learning state set consisted of only seven variables (Table 2.2).

Feature	Values	Explanation
Greeting	0,1	Whether the system has greeted the user
Attribute	1,2,3,4	Which attribute is being worked on
Confidence/ Confirmed	0,1,2,3,4	0,1,2 for low, medium, and high ASR confidence 3,4 for explicitly confirmed or disconfirmed
Value	0,1	Whether value has been obtained for current attribute
Tries	0,1,2	How many times current attribute has been asked
Grammar	0,1	Whether non-restrictive or restrictive grammar was used
History	0,1	Whether there was trouble on any previous attribute

Table 2.2: State features and values in the NJFun dialogue system.

The state space defined by these variables contained $2 \times 4 \times 5 \times 2 \times 3 \times 2 \times 2 = 960$ unique states. However, many of these states would not actually occur. For example, Greeting=0 was only possible in the initial dialogue state. All other states where Greeting=0 would never occur. Consequently, only 62 states could actually occur in a dialogue. Furthermore, by hard-coding obvious decision choices, the authors were able to reduce the number of significant states to just 42. The purpose of learning was to determine which was the most appropriate action in each of these 42 states.

Only two actions were made available in each of the 42 contentious states. The action alternatives were chosen so that the dialogue system could randomly choose an action and still appear to behave sensibly to a user. In many of the 42 states, the choice was between confirming an attribute value or not; the remaining choices centred on which type of initiative to use when greeting the user or asking a question (Table 2.3).

Action choice	Number of states
Confirm vs NoConfirm	30
ReAsk (system vs mixed initiative)	9
Ask (system vs user initiative)	2
Greet (system vs user initiative)	1
Total	42

Table 2.3: Summary of action choices in contentious states in NJFun.

Using this framework, the unlearned system was used by 54 human participants. Each participant was asked to complete six free-form tasks (e.g. "*You are bored at home in Morristown on a rainy afternoon. Use NJFun to find a museum to go to.*"). A total of 311 dialogues was completed. Whenever one of the 42 contentious states occurred in a dialogue, the system randomly chose between the two available actions. After completing each dialogue, users were asked to rate the success of the dialogue according to a number of user satisfaction measures. The logged data was then used to model the transition and reward functions. The transition function was based on the relative frequency of the actions chosen in each contentious state. Data sparsity was prevented by the small state set and the fact that only two actions were allowed in each state. The reward function was a simple binary one: a value of 1 was assigned at the end of the dialogue if the system queried the database using exactly the attributes specified in the task description, and -1 otherwise.

Using these transition and reward functions, dynamic programming was used to learn a strategy for the 42 contentious states. At the end of learning, a preferred action was identified in 36 of the 42 states. The new learned strategy was evaluated by a new group of 21 participants, performing the same six tasks as used in the previous evaluation. A total of 124 dialogues were completed. The evaluation results reported a number of positive findings including a statistically significant increase in the task completion rate from 52% to 64%. This study illustrates the potential benefit of modelling dialogue as a MDP. Other studies demonstrating similar benefits include Levin et al. (2000); Scheffler and Young (2002) and Lemon et al. (2006).

2.4.6 Dialogue simulation

In the case study just described, the creation of a realistic transition function was possible because the number of state transitions under consideration was very small. However, with larger state sets, modelling a transition function directly from a corpus would inevitably lead to an issue with data sparsity. Some studies in strategy learning have proposed methods, such as back-off schemes, to compensate for the sparseness of collected data (Goddeau and Pineau, 2000). However, directly modelling state transition probabilities gives rise to another, more serious problem. With this approach, the state and action sets need to be known in advance so that the corpus can be annotated for estimating the transition probabilities. If a system developer wishes

to experiment with an alternative state representation – and this often happens – the corpus will need to be re-annotated. This process is potentially very time-consuming.

One way to avoid both these problems is *dialogue simulation*. Simulated users were discussed briefly with regard to strategy evaluation (section 2.3.3). These models of user behaviour can also be employed to generate an unlimited number of training episodes for a strategy learner and, thus, allow a potentially exhaustive search for a useful strategy. More importantly, these models can be easily modified to accommodate alternative state and action sets. Consequently, simulated users have become an essential part of strategy learning techniques.

In recent years, a number of techniques have been employed in the development of simulated users. Typically, user behaviour is modelled at the level of dialogue acts (also referred to as 'intention level'). Some implementations have modelled behaviour at word level (Watanabe et al., 1998) or even acoustic level (López-Cózar et al., 2002). Whichever level of representation is chosen, simulated users normally include some model of the errors associated with speech recognition (hence, it is more accurate to refer to a simulated *environment*). Unfortunately, misrecognition remains a common occurrence in dialogue systems, so it is important to incorporate a model of recognition errors in dialogue simulations (Lin and Lee, 2001; Pietquin and Beaufort, 2005).

Early models of user behaviour used simple bigram models to generate the simulated user's response based solely on the system's last dialogue act (Eckert et al., 1998). Although this model was simple to devise and was task-independent, it did not take account of the user's long term goals. Consequently, it sometimes generated actions that did not make contextual sense. The notion of a *goal-directed* simulated user was introduced by Scheffler and Young (2000). In this work, the authors created a finite-state network of possible paths through a simulated dialogue. Some choice points were hand-coded, while others were trained from a corpus. This approach generated responses that were always consistent with a specified user goal. On the other hand, the approach was highly task-dependent.

Increasingly sophisticated probabilistic, goal-directed simulated users have been implemented in recent years. This has been achieved using a variety of representations and techniques, including: hand-coded probabilistic rules (Chung, 2004), Bayesian networks (Pietquin and Dutoit, 2006), n-grams with linear feature combination (Georgila et al., 2005) and Hidden Markov Models (Cuayáhuitl et al., 2005). For a

recent survey of dialogue simulation techniques, see Schatzmann et al. (2006).

It seems likely that the performance of any strategy learner depends, not only on the chosen state representation and reward function, but also on the quality of the simulated user(s) with which it interacts. Therefore, it is natural to ask what defines a good simulated user. In simple terms, a simulated user should be representative of the real users that will eventually interact with the learned strategy. A number of evaluation metrics were used to measure the realism of simulated users in Schatzmann et al. (2005). In this study, a comparative evaluation of three simulation methods was performed. The results indicated that "simple statistical metrics are still sufficient to discern synthetic from real dialogues" (p. 45).

2.4.7 Technical challenges

A great deal of progress has been made with the use of RL techniques to generate dialogue strategies. However, a number of important challenges remain. For instance, very little success has been achieved with the large state-action spaces that are typical of real-life systems (Henderson et al., 2005). Similarly, work on summarising learned strategies for interpretation by human developers has so far only been applied to tasks where each state-action pair is explicitly represented (Lecœuche, 2001). This tabular representation severely limits the practical size of the state space.

In addition, misrecognition on the part of the speech recogniser can create situations where the dialogue manager makes decisions based on an inaccurate assessment of the dialogue state. In other words, the true state of the dialogue is only partially observable by the dialogue manager. Recent work has incorporated a model of this partial observability into the dialogue strategy, albeit with relatively small state representations. Experimental results show that incorporating such a model into a strategy learning framework can produce superior strategies (Williams et al., 2005).

Eventually, this learning framework for strategy generation needs to be embodied in a software tool. Such a tool should require a system developer to specify only what kind of behaviour is required of the dialogue strategy and allow the learning framework to generate a strategy that fulfills these requirements. In the broadest sense, the tool should allow system developers to build good dialogue strategies quickly. More concretely, the tool should be able to:

i. generate strategies that compare favourably with hand-coded ones;
ii. generate strategies quickly (e.g. within hours instead of days or weeks);
iii. improve existing strategies where possible;
iv. handle a wide variety of tasks;
v. produce understandable results;
vi. automatically halt when a solution is found;
vii. incorporate domain-specific knowledge;
viii. compensate for the uncertainty associated with speech recognition;

Given the current state of the art, a software tool that satisfies all of these requirements may be some way off. There remain a number of important technical challenges to be tackled before all of these criteria can be met; some of these are discussed below.

2.4.7.1 Large state-action spaces

Bearing in mind that tabular RL algorithms, such as Q-learning, are designed to operate on individual state-action pairs, there is a practical limit to the size of the state-action tables that can be implemented. This constraint poses a problem for dialogue strategy developers. Even with a relatively small number of state features and system actions, the size of the state-action space can grow very quickly. In the context of reinforcement learning problems, "large" means a state-action space that would require greater than, say, one million (10^6) elements when represented in tabular form. Realistic state-action spaces can very easily reach or exceed this figure (e.g. six state variables with ten possible values). Large state-action spaces also increase the time taken to learn an effective strategy (Denecke et al., 2004). In summary, large state-action spaces are a major issue when learning dialogue strategies.

Several approaches to dealing with the problem of large state-action spaces have been proposed in recent years. One approach is based on the idea that not all state variables are relevant for learning a dialogue strategy. By carefully selecting a subset of the available state variables, the state-action space is reduced. If the right variables are chosen, useful dialogue strategies can still be learned. This technique has been applied

successfully in several recent studies, including Frampton and Lemon (2006), Tetreault and Litman (2006) and Reiser and Lemon (2006).

A related technique is to consider only valid state-action pairs when learning a dialogue strategy. In the fully enumerated set of possible states, many of the states will not make sense. For example, if the status of a slot is represented by two binary variables, *filled* and *confirmed*, the state '*filled=false, confirmed=true*' should not occur. Using this idea, the size of the 'feasible' state-action space can be greatly condensed. For example, the case study presented in section 2.4.5 stated that only 62 of a theoretically possible 960 states would actually occur in a real dialogue (Litman et al., 2000).

More recently, this idea has been formalised into an algorithm for automatically eliminating invalid state-action pairs (Cuayáhuitl et al., 2006a). Using basic domain knowledge, the *sapReduction* algorithm can dramatically reduce the size of the state-action space. Experimental examples demonstrated a 94% reduction in the size of the state-action space and 93% faster rate of convergence towards an appropriate strategy.

The techniques discussed so far are practical ways to reduce the size of the state-action space so as to make the use of tabular RL algorithms feasible. It may be the case that, even for fairly sophisticated dialogue tasks, a small state-action space is sufficient to learn appropriate dialogue strategies. However, there may be times when a system developer decides that a large state-action space *is* required. One solution to this problem has been proposed in Henderson et al. (2005). This work used *linear function approximation* (LFA) to represent a state space with theoretically 10^{87} unique elements. This technique exploited the commonality between neighbouring dialogue states to define a function which approximates the Q-value for all possible states. In this study, the LFA-based state representation was used in conjunction with a hybrid reinforcement/supervised learning technique. This technique was applied to the Communicator task (Walker et al., 2001) to learn dialogue strategies that outperformed state-of-the-art hand-coded ones (Lemon et al., 2006).

2.4.7.2 Strategy Inspectability

It is important for human developers to understand, and learn from, the strategies generated by machine learning methods. Furthermore, it would be very useful for system developers to have an easy way to modify learned dialogue strategies.

Therefore, some method is required for summarising the behaviour of a learned strategy. Without the ability to inspect a learned dialogue strategy, a system developer might find it difficult to deploy the strategy in the real world. Certainly, it seems unlikely that a commercial enterprise would deploy a SDS containing a learned dialogue strategy if its behaviour was not completely inspectable.

However, summarising learned dialogue strategies is difficult. When state-action spaces are small enough to represent in tabular form, RL algorithms such as Q-learning can be applied to generate a dialogue strategy. However, the representation of the learned dialogue strategy is extremely fine-grained; every possible state-action combination is specified. Unless the state-action table contains less than a few hundred elements, strategy inspectability is virtually impossible. What is required is some method for summarising which action is appropriate in which states.

One approach to this problem is the use of inductive logic programming (ILP) to generate a set of *if-then* rules which summarise the state-action space of a learned dialogue strategy (Lecœuche, 2001). In this study, the summarisation method is illustrated by reducing 42 state-action pairs to 24 if-then rules after learning has occurred. The paper also described some preliminary work integrating the rule-based representation into the learning algorithm. Experimental results suggested that this might help to accelerate convergence towards an appropriate strategy. Clearly, results with much larger state-action spaces, and greater compression, are needed to be genuinely useful. Moreover, it is not clear how this approach could be applied successfully to non-tabular state representations (Goddeau and Pineau, 2000; Henderson et al., 2005)

It is worth noting that inspectability has long been an issue with other machine learning techniques, particularly neural networks (Haykin, 1999). Because the ‘solution’ produced by a neural network is represented by a set of numerical weights, it exhibits very low readability. Some techniques have been developed to make solutions more transparent, notably *sensitivity analysis* (Oh and Lee, 1995). However, extracting learned knowledge from neural networks remains difficult. Despite this fact, neural networks have been employed in a large number of research-led and commercial applications (Berry and Linoff, 2004, Chapter 13). Over time, the same perspective may be adopted by developers of learned dialogue strategies. For the moment, however, the inspectability of dialogue strategies remains an important goal.

2.4.7.3 Simulated users

The use of simulated users was reviewed briefly in section 2.4.6. While it is true that the sophistication of simulated user models has increased greatly in the last few years, some important issues remain unresolved. Firstly, there have been very few evaluations of learned dialogue strategies with real users (e.g. Litman et al., 2000; Walker, 2000). Moreover, only one of these has learned a dialogue strategy using a simulated user and tested the learned strategy with real users (Lemon et al., 2006).

Certainly, conducting evaluations with real users is time-consuming and expensive. However, the quality of any dialogue strategy can only be properly assessed by human users. Therefore, it is important that more work be done to assess whether good results with simulated users are a reliable predictor of subsequent performance with real users.

Secondly, very little attention has been given to the cost in time and effort required to collect and annotate a corpus for the purpose of creating a simulated user. I argue that an implicit motivation for learning dialogue strategies is to reduce the time taken to develop SDSs. If novel applications are to be developed in the future, then no relevant corpora will exist. In this situation, a system developer is faced with the choice of hand-coding a simulated user or collecting a corpus. For complex tasks, hand-coding simulated users may not be feasible. However, I suggest that, for many information-seeking tasks, hand-coding simulated users *is* feasible (see Chapters 4 – 6). If this is not the case, an implicit benefit of learning dialogue strategies will be lost. Thus far, there has been no cost-benefit analysis of the hand-coded versus corpus-based approaches.

2.4.7.4 Partial observability

Modelling dialogue as a Markov Decision Process assumes that the state representation reflects all the information needed to decide on which action to take next. However, this is not actually the case. The errors associated with speech recognition mean that the user's intentions may not be accurately communicated to the dialogue manager. Consequently, the dialogue state may not represent the true state of the dialogue, from the user's perspective. In terms of modelling dialogue as a MDP, the true dialogue state is said to be *partially observable* by the dialogue manager.

In an information-seeking (slot-filling) task, the usual approach to dealing with speech recognition errors is to associate a recognition confidence score with each slot. In this type of task, the value of each slot is usually considered to be correct (i.e. what the user actually said) if either the slot value has a high confidence score or the slot value has been confirmed by the user in some way. Although a simplification of what happens in slot-filling tasks, this description is a reasonable approximation. Studies of more sophisticated confirmation strategies include Raymond et al. (2003) and Litman and Hirschberg (2006). The recognition confidence scores are received by the dialogue manager from the speech recogniser for each user utterance.

In recent years, the MDP approach to learning dialogue strategies has been extended to include a model of the errors associated with speech recognition. Since this model takes account of the inherent partial observability, it is known as a Partially Observable Markov Decision Processes (POMDP). The basis of the POMDP approach is the use of a *belief state*, which is a probability distribution over the set of all possible dialogue states. In other words, multiple dialogue state hypotheses are maintained at each point in the dialogue. The idea is to update the probability for each possible state that it is the true state as the dialogue progresses. Using this extended state representation, several POMDP algorithms exist for generating optimal policies (e.g. Kaelbling et al., 1998). However, the computational complexity of these algorithms is exponential in the number of states. This makes learning a dialogue strategy intractable with anything more than a handful of states. Therefore, methods of approximating the belief state or finding near-optimal policies are used to make learning computationally feasible. This approach has been shown to improve the performance of a SDS in Roy et al. (2000). The study also showed that the margin by which the POMDP-based strategy outperforms the MDP-based one, increased with the level of recognition error in the SDS.

The most comprehensive study of the POMDP-based approach to dialogue strategy learning can be found in Williams (2006). In this work, the state representation differs from the approach normally found in MDP-based state spaces. Slots are not characterised by their status (e.g. empty, filled, confirmed etc.) but by their possible values (e.g. ‘London’, ‘Edinburgh’, ‘Cambridge’ etc.). This is because the main point of the POMDP approach is to explicitly model multiple state hypotheses. The work presents a novel technique, Composite Summary Point-Based Value Iteration (CSPBVI), to tackle the issue of scalability.

The author also describes a working SDS, which includes a dialogue manager based on the CSPBVI algorithm. The system is a simple ticket booking application. However, the size of the state space is $3 \times 10 \times 7 \times 9 = 1890$ unique slot value combinations ("user goals"). Clearly, many applications would require a much larger slot value space. Nevertheless, the work clearly demonstrates that the POMDP approach can model the effect of recognition errors and consequently outperform MDP-based dialogue strategies.

2.5 Summary

A spoken dialogue system is a natural language interface designed to make use of spoken language technology for both language understanding and generation and so to effect a conversation between a user and computer. Such a system usually provides access to a computer-based application such as a database or an expert system. From a system developer's perspective, designing a spoken dialogue system can be a time-consuming and difficult process. A developer may spend a lot of time anticipating how a potential user might interact with the system and then deciding on the most appropriate system response. These decisions are encoded in a *dialogue strategy*.

To reduce the time and effort associated with developing a dialogue strategy, recent work has focused on generating strategies automatically. The idea is for developers to specify *what* behaviour is required of the system rather than *how* it is to be achieved. That is, the dialogue strategy is automatically generated from the developer's specification. One approach to accomplishing this is to model a human-computer dialogue as a Markov Decision Process (MDP). MDPs are based on the notion that the current state of the environment contains all the relevant information needed to decide on which action to take next. This idea is akin to the concept of *dialogue state*. In many spoken dialogue systems, the dialogue state serves as a compact representation of the dialogue's history. By formalising a dialogue as a MDP, reinforcement learning (RL) algorithms can be used to search for dialogue strategies. Such strategies are often learned by repeated interaction with a simulated user. This 'user' is a software-based model of how a human user might interact with a spoken dialogue system.

A great deal of progress has been made with the use of RL techniques to generate dialogue strategies. However, a number of important challenges remain. For instance, very little success has been achieved with the large dialogue state representations that are typical of real-life systems. Similarly, work on summarising learned strategies for interpretation by human developers has so far only been applied to very simple dialogue strategies. Also, it is not yet clear whether learning dialogue strategies with simulated users provide a reliable predictor of subsequent performance with real users. Finally, it has been shown that incorporating a model of the errors associated with speech recognition can improve the quality of a learned dialogue strategy. However, more work is required to exploit these error models with large state representations.

Chapter 3

An evolutionary approach to learning spoken dialogue strategies

3.1 Overview

In this chapter, I present an alternative to existing approaches for learning spoken dialogue strategies. I propose the use of an evolutionary-based algorithm, XCS (Wilson, 1995). This algorithm has been applied successfully to a number of supervised learning and reinforcement learning tasks. The principal advantage of XCS over tabular TD algorithms, such as Q-learning, is the ability to represent *regions* of the state-action spaces rather than single points (i.e. individual state-action pairs). This means that much larger state-action spaces can be represented, which in turn means that more realistic problems can be tackled.

Before describing the XCS algorithm, I present some background on evolutionary algorithms (EAs) and show how they have been applied to sequential decision tasks. Next, I introduce Learning Classifier Systems (LCS), a well known example of a rule-based EA. I then present XCS, an algorithm that combines an evolutionary algorithm with a Q-learning update rule, to generate condition-action rules that can compactly represent solutions to sequential decision tasks. I describe *classifiers*, the basis for knowledge representation in XCS and then describe how XCS evolves a population of classifiers that accurately describes a solution to a sequential task. Finally, I outline how I will use the XCS algorithm to learn spoken dialogue strategies.

3.2 Evolutionary Algorithms

Evolutionary Algorithms (EAs) are a group of stochastic search techniques inspired by Darwin's theory of evolution by natural selection. EAs operate on a population of potential solutions, often encoded in structures called *chromosomes*. This population of chromosomes is iteratively modified using operators borrowed from genetics. The idea is that successive generations of the population contain chromosomes that are better suited to their environment than the chromosomes from which they were created, just as in natural selection. In other words, chromosomes which encode the best solutions evolve over time and eventually dominate the population in a way that is similar to the notion of 'survival of the fittest'. This remarkably simple process of combining candidate solutions to produce progressively better solutions has proven to be very powerful. Figure 3.1 shows the structure of a generic evolutionary algorithm.

```
1   randomly initialise population with n chromosomes
2   repeat
3      evaluate the fitness of each chromosome
4      select parents stochastically according to their fitness
5      generate n offspring by transforming parents
6      replace the old population with the n offspring
7   until termination criteria met
```

Figure 3.1: A generic evolutionary algorithm.

The term 'Evolutionary Algorithm' has been coined to collectively describe four different approaches that have been developed independently over the last 40 years: Evolutionary Strategies (Rechenberg, 1964), Evolutionary Programming (Fogel et al., 1966), Genetic Algorithms (Holland, 1975) and Genetic Programming (Koza, 1992). Each of these approaches incorporate the notion of a *population-based* search for solutions. However, there are some important differences. For example, in Genetic Algorithms (GAs), solutions are often encoded as fixed length strings of bits, integers or real values. In Genetic Programming (GP), each chromosome is a variable length tree structure representing part of a computer program; in Evolutionary Programming, solutions are represented by finite state machines. Another difference relates to how chromosomes are transformed. In GAs, the transformation is primarily accomplished by the crossover operator. This operator produces new offspring chromosomes by

combining the information contained in its parental chromosomes. By contrast, the other forms of EA have traditionally made more use of the mutation operator, where parts of a chromosome are randomly modified. As variations in the basic forms of EA have been developed over the years, it has become clear that the methods have merged to some degree. For instance, the role of mutation has become more important in the design of GAs. Similarly, the crossover operator is now more prevalent in Evolutionary Strategies.

The XCS algorithm presented later in this chapter has features that are most closely associated with GAs. These features include: (i) fixed length chromosomes; (ii) selection of chromosomes for reproduction in proportion to their fitness; (iii) reproduction of chromosomes by genetic operators (crossover and mutation).

3.2.1 Evolutionary Algorithms for sequential decision tasks

EAs have been applied successfully to a diverse range of learning, classification and optimisation problems (Goldberg, 1989; Obayashi et al., 2000; Andreou et al., 2002; Koza et al., 2003). EAs have also been employed to search for policies in sequential decision tasks (Moriarty et al., 1999). This approach differs from the one presented in section 2.4. In that section, I discussed how reinforcement algorithms, such as Q-learning, can be used to find an optimal action-value function and from this function, a policy can be induced. By contrast, the EA approach is a form of *direct policy search*. An optimal policy is derived by searching the policy space directly rather than searching the value space and from that deriving a policy. In other words, an initial estimate π_0 of the optimal policy π^* is iteratively refined $(\pi_1, \pi_2, \ldots, \pi_t)$ to produce increasingly more accurate estimates of π^*. It has been suggested that a direct policy search may have advantages over a search for an optimal action-value function (Baxter and Bartlett, 2001; Rosenstein and Barto, 2001). A brief discussion of the merits of policy search versus value function search is given later in section 3.2.5.

To explain how EAs (particularly the GA form) can be employed to generate optimal policies for sequential decision tasks, I will address the following questions: (i) how are policies represented? (ii) how are policies evaluated and selected for transformation? (iii) how are policies transformed to produce better ones?

3.2.2 Policy representation

In the context of GAs, a *gene* is a subsection of a chromosome which usually encodes the value of a single parameter. Thus, the simplest way to represent a policy in a GA is to use a single chromosome per policy with a single gene associated with each state. Each chromosome comprises a list of observed states and a preferred action for each state (Figure 3.2a). This is the same type of policy as would be derived using tabular RL algorithms such as Q-learning. However, for many applications the number of observable states is too large, making this method of policy representation impractical (just as with tabular RL).

Consequently, more compact representations of policies are often required. One approach is to replace the state-action table with a sequence of weights for a neural network (Figure 3.2b). The neural network architecture represented by these weights acts as a function approximator. That is, the neural net approximates the policy in a compact way, accepting the state representation as input and generating the optimal action as output. Example applications of this approach include Whitley et al. (1993) and Yamauchi and Beer (1993). Another approach is to represent a policy as a set of condition-action rules (Figure 3.2c). Each condition expresses a predicate that matches a set of states. In this way, a condition identifies a region of the state space rather than a single state. Examples of this approach include Grefenstette et al. (1993) and Smith et al. (2000).

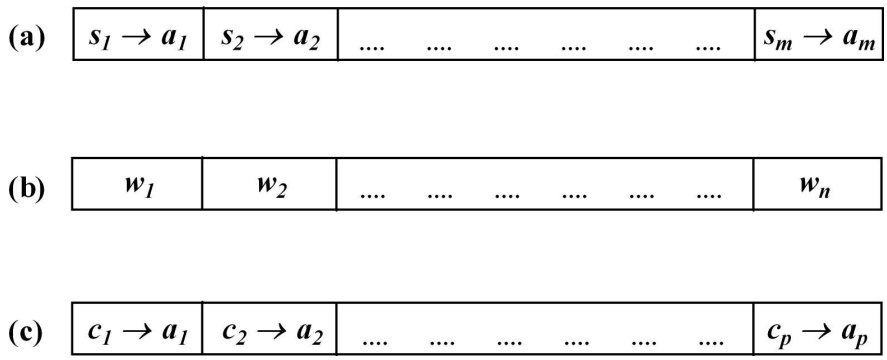

Figure 3.2: Schemes for representing policies within a single chromosome: (a) state-action table; (b) neural network weights; (c) condition-action rules. [a]

[a] In practical applications, the number of weights and condition-action rules are very much less than the number of states (i.e. $n, p \ll m$).

A further refinement is to distribute the representation of a policy over several chromosomes, or even over the entire population of chromosomes. In other words, rather than representing an entire policy, each chromosome expresses only part of a policy. This approach has two possible advantages. Firstly, it allows learning to work on a more detailed level of the task. It seems likely that evolving a policy for a subtask is easier than evolving a policy for the complete task. Consequently, a complete policy can be represented as a set of 'sub-policies'. Secondly, decomposition facilitates the exploitation of domain knowledge. For example, a system developer might separate a task into components that can be hard-coded while other components can be learned. An example of a distributed neural network-based application is the SANE system (Moriarty and Miikkulainen, 1998) whereas, the XCS algorithm – described in Section 3.4 – is an example of the distributed rule-based approach.

Finally, we must address the issue of how policies are actually encoded. Most GA applications use fixed-length, fixed-order bit strings to encode solutions. For example, in a very simple robot navigation problem, four actions could be encoded as two bits: "step back" (00), "turn right" (01), "turn left" (10), "step forward" (11). A complete policy for an environment containing only six states might be: $\boxed{00\ 10\ 11\ 10\ 11\ 11}$. This chromosome indicates that the robot should step back in state 1, turn left in states 2 and 4 and step forward in all other states.

There are a number of reasons for the prevalence of binary encoding of chromosomes in GAs. The original work on GAs by John Holland used this encoding; thus, much of the existing GA theory was developed in the context of binary encodings. Furthermore, heuristics concerning appropriate parameter settings have generally been developed with binary encodings. However, for many applications, binary encoding may be unsuitable. For example, neural network weights are normally best encoded by real values (Montana and Davis, 1989). In applications with many actions, it might be more appropriate to employ more compact, character-based encoding schemes (e.g. {B, L, R, F} for the robot navigation example above). Selecting an appropriate encoding scheme can often be crucial to the success of a GA-based algorithm and a wide range of schemes have been devised over the years. Examples of alternative encoding schemes and discussions of their effectiveness can be found in: Goldberg (1989, pp.80–82), Mitchell (1996, pp.156–166) and Salomon (1996).

3.2.3 Fitness function and selection

In order to produce successively better policies from one population to the next, some method of evaluating the performance of individual polices is required. This is achieved using a *fitness function*. The role of the fitness function is to define the criterion for ranking potential solutions and for stochastically selecting them for inclusion in the next generation of the population. The form of the fitness function depends on the type of solution being represented. In a classification task, for example, the fitness function typically scores the classification accuracy of each solution over a set of training data.

When the solution is a policy, the fitness function should reflect the reward that would be accumulated by an agent employing that policy in a specified environment. In a deterministic environment, a policy's fitness can be evaluated in a single episode (i.e. the sequence of steps from the initial to the terminal state, see Section 2.4.1). In a stochastic environment, the fitness is normally averaged over a number of episodes. Calculating the fitness of policies can become computationally expensive. In fact, fitness scoring can sometimes account for a high proportion of the total computational effort. In tasks where there are many steps in a single episode, alternative fitness measures can be employed, including: (i) the total reward achieved after a fixed number of steps; and (ii) the number of steps required to attain a fixed level of reward.

When a fitness score has been assigned to each candidate policy, some method of selection is required for deciding which chromosomes will be included in the next generation of the population. A commonly used selection method is to probabilistically select chromosomes based on relative fitness:

$$P(p_i) = \frac{Fitness(p_i)}{\sum_{j=1}^{n} Fitness(p_j)} \tag{3.1}$$

where p_i represents policy i and n is the total number of policies. This method is sometimes referred to as fitness-proportionate selection, or roulette-wheel selection. In another method, rank selection, the policies are first sorted by fitness. The probability that a policy is selected for inclusion in the next generation is proportional to its rank rather than its fitness. This method can prevent highly fit solutions from dominating the population too quickly. A variety of other selection methods have been proposed, including tournament selection, scaling selection and others (Goldberg and Deb, 1991).

3.2.4 Genetic operators

Once the fittest chromosomes have been selected, they must be transformed with a view to producing even fitter chromosomes in the next generation. There are two basic methods for achieving this. The first is the *crossover* operator, which produces two new offspring from two parent chromosomes by copying selected genes from each parent. The simplest form of crossover is single-point crossover, where a crossover point is chosen at random and the parts of each parent after this point are exchanged to form the offspring (Figure 3.3a). Other forms of crossover include multi-point (De Jong and Spears, 1992) and uniform (Syswerda, 1989).

The second method of transformation is the *mutation* operator (Figure 3.3b). This operator produces small random changes to a chromosome by modifying the value of each gene with a small probability. Consequently, for a particular chromosome, none, one, or more genes may mutate in a single operation. Mutation is often performed after crossover has been applied.

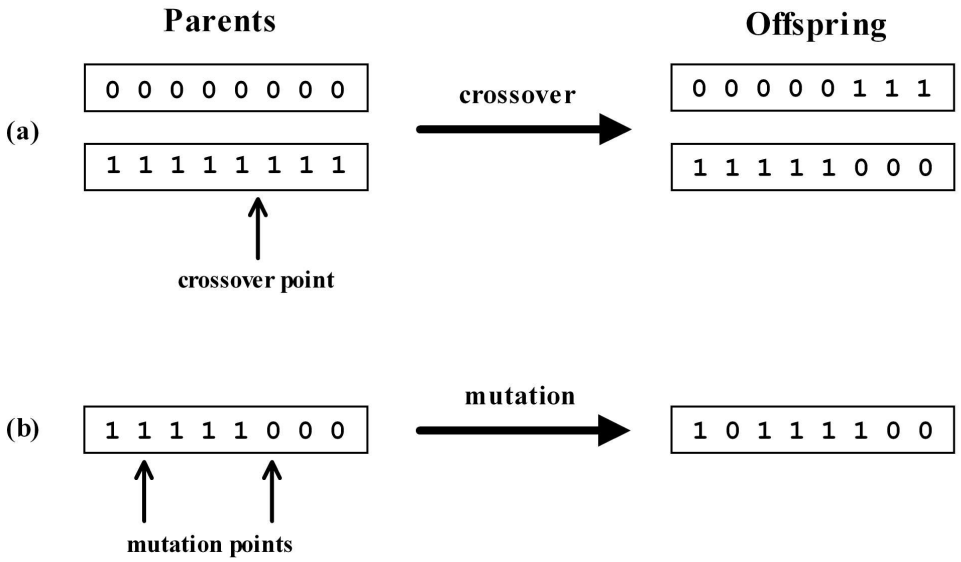

Figure 3.3: Simple genetic operators: (a) single-point crossover; (b) mutation.

The idea behind crossover is that fit solutions consist of building blocks (*schemas*) that can be recombined to produce even fitter solutions (Mitchell, 1996, p.27). In other words, crossover seeks to exploit promising regions of the search space.

Mutation, on the other hand, serves to preserve population diversity and thus facilitate occasional exploration of new regions in the search space. Empirical results over the years have suggested that a GA should use a high probability of crossover (e.g. 0.9) and low probability of mutation (e.g. 0.01). Typically, the mutation probability is selected so that, on average, one gene per chromosome is mutated. Since these operators are implemented in a probabilistic fashion, occasionally a chromosome will simply be cloned from one generation to the next (i.e. neither crossover nor mutation occurred). More detailed descriptions of crossover, mutation and other genetic operators, together with a discussion of empirical results can be found in Goldberg (1989, Chapter 5) and Mitchell (1996, Chapter 5).

3.2.5 Strengths and limitations

It has been argued that EA-based policy search techniques enjoy several advantages over tabular value function methods (Moriarty et al., 1999; Schmidhuber, 2000). Firstly, most policy search methods specify the policy at a higher level of abstraction than an explicit state-action mapping. Policy-based methods generally facilitate some form of state aggregation, where states that require the same action are grouped together (e.g. as a condition-action rule). Moreover, value function methods include many state-action pairs where the expected reward is very low. In other words, a record of all possible decisions is kept, including many bad ones. The purpose of policy search methods is to represent only the good decisions. Actions that are sub-optimal for certain states will have a low fitness score and so will eventually be eliminated from the population. Both of these points mean that policy search methods facilitate more compact state-action representations. Consequently, problems requiring very large state-action spaces can be handled.

On the other hand, the aggregation of states and tendency to eliminate actions with low reward can be a disadvantage with respect to rarely occurring states. In single-chromosome policy representations, genes representing rarely occurring states will have little impact on the fitness of the policy as a whole. As a result, random mutations will tend to accumulate in these genes. In distributed policy representations, chromosomes representing rare conditions (state sets) are unlikely to be selected for recombination and will eventually be deleted from the population. Therefore, EA-based methods of policy search often perform poorly with respect to rarely occurring

states. In tabular value function methods, the value of every state-action pair is kept, and so information about rarely occurring states is not lost. Naturally, the value estimates for actions in these states will tend to be much less accurate than those of frequently occurring states.

In sequential decision tasks, value function methods perform optimally when the state of the environment is fully observable by the learning agent's sensors. That is, the state representation of the environment is Markov. However, in many realistic tasks, the Markov Property does not hold; the true state of the environment is often only *partially* observable. For example, in a robot navigation task, limitations in the robot's sensors may mean that different real world states (e.g. locations in a building) may sometimes be indistinguishable by the robot (Crook and Hayes, 2003a). States that are different in the real world but perceived to be the same are known as *perceptually aliased*[1]. A similar problem occurs in spoken dialogue systems. The limitations of the speech recogniser can create situations where a user's intentions are misunderstood by the dialogue manager.

Policy search methods, however, make no assumptions about the observability of the environment. Reward is associated with entire policies (or sub-policies) rather than individual states. Put another way, value-based methods reflect the structure of the problem whereas policy-based methods reflect the structure of the solution (Rosenstein and Barto, 2001). By taking a broader view of how to assign reward, policy-based methods are less vulnerable to the effect of partial observability.

Finally, it is important to remember that the arguments presented in this section relate to the differences between EA-based policy search and value function search where the value function is tabular (i.e. all state-action mappings are represented). In cases where the value function is represented in a generalised form (e.g. using a neural network as a function approximator), the same strengths and limitations of policy search methods may apply. Thus far, there appears to have been no direct comparison of value space and policy space techniques methods where generalisation is employed in both methods. A more detailed discussion of the strengths and limitations of EAs for policy search (including illustrative examples) is given in Moriarty et al. (1999).

[1]There are situations when augmenting an agent's sensors will still not prevent the problem of perceptual aliasing. For example, two different corridors in a building may be appear identical, even to humans. One way to make the environment fully observable is to give the learning agent some facility to remember where it came from (e.g. Lanzi and Wilson, 2000).

3.3 Learning Classifier Systems

A well known example of a rule-based EA that can be applied to sequential decision tasks is the Learning Classifier System (LCS) model. This model was introduced by John Holland in the 1970s as a framework for learning rule-based knowledge representations (Holland, 1976). In this model, a rule base consists of a set of N condition-action rules known as *classifiers*. The condition part of a classifier is represented by a ternary string from the set $\{0,1,\#\}$ while the action part is composed from $\{0,1\}$. The # symbol acts as a wildcard allowing a condition to represent a set of states. For example, the condition string 1#1 matches the states 111 and 101.

Classifier systems search for optimal rule sets by combining two machine learning techniques. A genetic algorithm is used to evaluate and modify the population of rules while reinforcement learning is used to assign rewards to existing rules. The search for better rules is guided by the *strength* parameter associated with each classifier. This parameter serves as a fitness score for the genetic algorithm and as a predictor of future reward for the RL component. This evolutionary learning process searches the space of possible rule sets to find an optimal one.

Both the single-chromosome and distributed forms of policy representation are available within the LCS paradigm. In *Pittsburgh*-style systems (Smith, 1983), a set of classifiers, consituting a complete policy, is represented within a single chromosome. In a *Michigan*-style system (Holland and Reitman, 1978), the policy is distributed over the entire population, with each chromosome representing only a single rule (i.e. a classifier). The names of these approaches refer to the univeristies where they were first developed. Pittsburgh-style systems are generally able to solve more complex problems but are very computationally intensive and slow to find solutions (Wilcox, 1995). As a result, Michigan-style systems are more commonly used. Some hybrid Pittsburgh-Michigan models have been developed in an attempt to retain the advantages of both approaches (Ishibuchi and Nakashima, 2000).

Finally, it should be noted that LCS are also well suited to supervised learning tasks. For instance, in data mining problems, the optimal rule set can represent classification rules, rather than a policy. In fact, many of the practical applications of LCS have occurred in this field (Lanzi et al., 2000; Bull, 2004; Bull and Lanzi, 2007). In this thesis, however, I am concerned only with the solving of sequential decision problems.

3.4 XCS

XCS (eXtended Classifier System) is a Michigan-style (i.e. distributed) classifier system that incorporates a number of modifications to Holland's original LCS framework. Firstly, the GA is not applied to the entire population for the creation of the next generation. Instead, only the rules which match the current input and take the selected action, are modified by the GA. Secondly, and more importantly, the strength parameter associated with each rule (classifier) is no longer used as an indicator of a rule's fitness. Instead, a classifier's fitness is based on the *accuracy* of the predicted reward rather than the reward itself. These two modifications result in a strong tendency by XCS to evolve accurate, maximally general classifiers that efficiently cover the state-action space of the problem (Wilson, 1995). An explanation of how this is achieved is presented in the subsequent sections.

The primary motivation for these modifications was the desire to build complete but compact maps of the state-action space. In common with many EA-based policy search methods, traditional LCS tended to delete conditions that rarely occurred. Recall that an advantage of value function methods was that they maintain a complete map of the state-action space. The tendency of XCS to evolve accurate, complete state-action maps make it more akin to Q-learning than policy search. At the same time, XCS retains the advantage of a generalised state representation – the ability to represent large state spaces. In fact, it has been shown that XCS is a proper generalisation of tabular Q-learning (section 3.4.4). The XCS algorithm has proven suitable for a wide range of supervised learning and reinforcement learning tasks, where generalisation over states is desirable (Wilson, 2001; Lanzi et al., 2005).

The sections that follow (3.4.1 to 3.4.4) provide a brief overview of XCS, with reference to sequential decision tasks. Since the algorithm was first proposed in 1995, it has become the best known and well researched form of LCS. Consequently, there is now a substantial, and growing, body of literature on the algorithm. Several of the most readable accounts of XCS are by the algorithm's creator, Stewart Wilson, including: Wilson (1995) and Wilson (1998). An excellent pseudocode description of the algorithm and suggestions for parameter settings are given in Butz and Wilson (2002). Finally, a comprehensive study of the algorithm, covering many aspects, but particularly the theory that underpins the accuracy-based fitness, is Kovacs (2002).

3.4.1 Classifiers

Before describing how the XCS algorithm can be used to learn policies, it is important to describe the basic unit of XCS, and LCS in general – the classifier. In a traditional Michigan-style LCS, a policy is represented by a set of classifiers with each classifier comprising a {*condition, action, strength*} tuple. The strength parameter defines the expected reward associated with selecting the specified action in the specified condition. For example, the three rules (classifiers) shown below represent a policy for an agent to choose between two actions (1 and 2) in eight possible states (represented as binary strings).

	condition	**action**	**strength**
a	010	1	200
b	111	2	300
c	###	1	50

Since rule *c* consists entirely of *wildcards* (also known as "don't-care" symbols), it is the most general rule possible. Hence, there is an overlap between it and the other two rules. For rule *a*, this is not a problem, because both rules *a* and *c* advocate the selection of action 1. But what happens if the agent enters state "111"? Because classifier *b* has a higher strength, action 2 will be selected. In each of the remaining six possible states, only the condition of classifier *c* will match, in which case action 1 will be selected.

Recall that the strength parameter has a dual purpose in LCS. It defines both the expected reward for action selection *and* the fitness measure for the genetic algorithm. In XCS, strength is replaced by three parameters: prediction, prediction error and fitness.

	condition	**action**	**prediction**	**pred. error**	**fitness**
a	010	1	200	0.1	725
b	111	2	300	0.0	999
c	###	1	50	0.8	34

Prediction assumes the role of the strength parameter for action selection. The prediction error and fitness parameters are inversely related and indicate the accuracy of the prediction. Calculating the accuracy of each classifier's prediction is the single most important feature of the XCS algorithm. A description of how these parameters are calculated is presented shortly (section 3.4.2.2).

XCS also introduces the notion of *macroclassifiers*, which is simply a classifier with an additional *numerosity* parameter. Whenever XCS generates a new classifier, the population is scanned to see if the new classifier has the same condition and action as any existing macroclassifier. If this is the case, the classifier is not added to the population. Instead, the existing macroclassifier's numerosity parameter is incremented by one. The existing macroclassifier's prediction, prediction error and fitness parameters are left unchanged. If instead, there is no existing classifier with the same condition and action, the newly generated classifier is added to the population and its numerosity parameter is given the value 1. For example, if we wish to insert into the existing population presented above, a new classifier for the condition-action pair $\langle 010, 1 \rangle$, we simply increase the numerosity of the existing macroclassifier *a* that has the same condition and action, thus:

	condition	**action**	**prediction**	**pred. error**	**fitness**	**numerosity**
a	010	1	200	0.1	725	2
b	111	2	300	0.0	999	1
c	###	1	50	0.8	34	1

Similarly, if a classifier is to be deleted from the population, its macroclassifier's numerosity value is decremented by one, unless its value is already 1, in which case the macroclassifier is completely removed from the population. Within the XCS literature, a macroclassifier is said to represent multiple copies of the same *microclassifier*. The implementation of macroclassifiers is essentially a programming technique to speed up the searching of classifiers and to reduce the size of the classifier population.

3.4.2 The XCS algorithm

I now present an overview of how the XCS algorithm works in the context of a sequential decision task (Figure 3.4). Before the algorithm begins its interaction with the environment, its population of classifiers [P] is initialised. Some example classifiers are shown in the figure; each classifier comprises a condition-action rule and the three parameters: prediction p, prediction error ε and fitness F. The population has a fixed maximum size N and may be initialised as: (i) an empty population (rules are added as required); (ii) N random rules; or (iii) N maximally general rules (i.e. all #'s). The basic steps in each cycle of the algorithm's execution are:

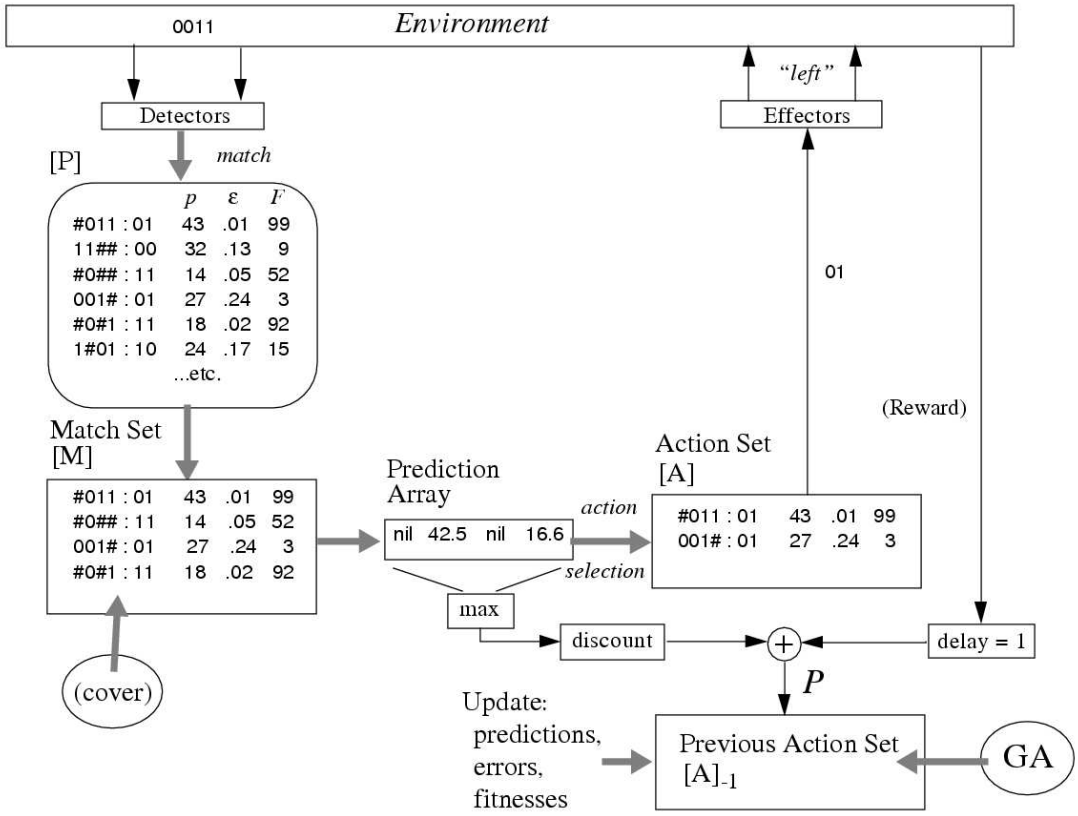

Figure 3.4: Schematic illustration of XCS (reproduced from Wilson (1998) by kind permission of Stewart Wilson).

1. the current state of the environment is received as input (from the "Detectors");
2. classifiers that match the current state form the *Match Set*; if an insufficient number of classifiers match the current state, then a new classifier is generated (using *covering*);
3. the Match Set is translated into the *Prediction Array* which reflects the average fitness-weighted prediction of each action in [M];
4. one action is chosen from the Prediction Array for execution;
5. all classifiers in the Match Set that propose this action form the *Action Set* [A];
6. the winning action is executed (by the "Effectors");
7. a reward *r* for the action just executed is received from the environment;
8. the *Previous Action Set* $[A]_{-1}$ is modified by the *payoff P* which is the sum of the previous cycle's reward and the discounted maximum prediction in [PA];
9. the GA is periodically applied to $[A]_{-1}$; two classifiers with high fitness are combined to produce offspring that replace two classifiers with low fitness.

This execution cycle continues until some termination criterion is met. The end result of this process is a classifier population that accurately predicts the expected reward for selecting actions in different regions of the state space. The algorithm will now be described in more detail in terms of its three main components:

- **Performance**: selecting actions based on the state of the environment;
- **Reinforcement**: rewarding classifiers that generate good performance;
- **Rule Discovery**: searching for classifiers that improve performance further.

3.4.2.1 Performance

The purpose of the Performance component is to select and execute actions. At each discrete time step, the current state of the environment is received as input. All classifiers within the current population that match the current state are known as the *Match Set* [M]. For example, in Figure 3.4 the state 0011 is matched by the classifier conditions: #011, #0##, 001#, #0#1. Since, the members of [M] may propose different actions, we need some method to determine which of the alternative actions is to be executed. To do this, each unique action in [M] and its associated *system prediction* is stored in a *Prediction Array* [PA]. For each action a_i in [PA], the system prediction $P(a_i)$ is calculated as a fitness-weighted average of the predictions of classifiers in [M] that advocate a_i.

Using this array, XCS can select the action to be executed in a number of ways: (i) deterministically select the action with the highest fitness; (ii) probabilistically select an action according to its relative fitness; or (iii) randomly select any one of the available actions. All classifiers in [M] that advocate the selected action now form the *Action Set* [A]. Finally, the selected action is sent to the environment and a reward r is returned to the system.

By selecting actions with a high fitness, the algorithm is seeking to gain the maximum reward. In a random selection, or selection of an action with low fitness, the algorithm executes a sub-optimal rule in order to explore new regions of the solution space. In XCS, the balance between exploitation and exploration is controlled by the parameter P_{explr}, which defines the probability of using random action selection in a given cycle.

3.4.2.2 Reinforcement

The purpose of the Reinforcement component is to reward classifiers that generate good performance. In sequential decision tasks, this is achieved by updating the p, ε and F parameters of each classifier in $[A]_{-1}$, the action set that was active in the last execution cycle.[2] The order in which the three parameters are updated can be varied. In this section, I will present the commonly adopted order: p, ε, F (Butz and Wilson, 2002). The update procedure begins with a calculation of the payoff P. This is the sum of the previous time step's reward r_{t-1} and the discounted value of the highest prediction in [PA] in the current time step:

$$P = r_{t-1} + \gamma \max_i P(a_i) \tag{3.2}$$

where γ ($0 < \gamma \leq 1$) is the *discount factor*. The prediction p of each classifier in $[A]_{-1}$ is updated by:

$$p \leftarrow p + \beta(P - p) \tag{3.3}$$

where β ($0 < \beta \leq 1$) is the *learning rate*. In RL systems, the learning rate (denoted α in the RL literature) is normally decayed over time. In XCS, however, the learning rate is normally constant. Next, the prediction error ε of each classifier is updated:

$$\varepsilon \leftarrow \varepsilon + \beta(|P - p| - \varepsilon) \tag{3.4}$$

Next, the fitness F of each classier is updated. To do this, we first compute the *accuracy* κ and *relative accuracy* κ' of each classifier in $[A]_{-1}$:

$$\kappa = \begin{cases} 1 & \text{if } \varepsilon < \varepsilon_0 \\ \alpha \, (\frac{\varepsilon}{\varepsilon_0})^{-\nu} & \text{otherwise} \end{cases} \tag{3.5}$$

$$\kappa' = \frac{\kappa}{\sum_{x \in [A]_{-1}} \kappa_x} \tag{3.6}$$

where ε_0 ($\varepsilon_0 > 0$) is the *accuracy criterion*. If the predicted error ε is below the threshold ε_0, the classifier is said to be *accurate* (i.e. $\kappa = 1$). Otherwise, the accuracy drops off quickly, depending on the values of α and ν.

[2] If this is the first step in a sequential task, the classifiers in the action set [A] are updated instead.

Finally, the fitness F of each classifier is updated towards its relative accuracy:

$$F \leftarrow F + \beta(\kappa' - F) \tag{3.7}$$

The net effect of these updates is to improve the accuracy of each classifier's prediction of reward. More specifically, each classifier's fitness F is an estimate of the classifier's accuracy, relative to other classifiers that typically occur in its action sets. This measure of accuracy facilitates a *selective pressure* on the genetic algorithm to evolve accurate, maximally general classifiers (Butz et al., 2001).

3.4.2.3 Rule Discovery

The purpose of the Rule Discovery component is to search for classifiers that improve performance further. It achieves this by introducing new classifiers into the population and removing old ones. Classifiers may be created on three different occasions: (i) by population initialisation; (ii) by covering; and (iii) by the genetic algorithm (GA). Covering occurs when the number of classifiers that match the current state is less than θ_{mna}. Typically, this parameter is set to the number of possible actions. When a classifier is created by covering, its condition is set to the current state of the environment and its action value is chosen randomly. Each bit in the condition is then mutated into a # with probability $P_{\#}$. Thus, a more general rule, that still matches the current state, is created.

Covering normally only occurs at the start of a learning episode. The vast majority of rule discovery takes place within the GA. The GA is applied to the Previous Action Set $[A]_{-1}$ if the average time since the last GA application to the classifiers in $[A]_{-1}$ exceeds a threshold θ_{GA}. The GA is applied as follows: (i) two classifiers are chosen with probability in proportion to their fitness; (ii) crossover is applied to the two classifiers with probability χ to create two offspring; (iii) mutation is applied to each bit in the offspring with probability μ; (iv) the two offspring are inserted into the classifier population [P]. The effect of this procedure is that accurate classifiers tend to reproduce and eventually dominate the population of rules.

If the population size N is exceeded when new classifiers are introduced into the population, some classifiers must be deleted. The process for accomplishing this, and other aspects of the Rule Discovery mechanism, are described in Kovacs (2002).

3.4.3 Parameterisation

The performance of XCS is controlled by the parameter settings listed in Table 3.1. Many of these parameters were introduced in the previous sections. In many applications of XCS, the value of many parameters are varied infrequently. In fact, Lanzi (2006) reported that "practically speaking, many of the parameters could be regarded as system constants". More details on these parameters, and a brief discussion of commonly used parameter settings is provided in Butz and Wilson (2002).

N	maximum size of the population (in microclassifiers)
β	*learning rate*
γ	*discount factor* (for updating classifier predictions)
α	*accuracy falloff rate* (for classifier fitness calculation)
ε_0	*accuracy criterion* (for classifier fitness calculation)
ν	*accuracy exponent* (for classifier fitness calculation)
θ_{GA}	GA threshold (for deciding when to apply the GA)
χ	probability of applying crossover in the GA
μ	probability of applying mutation on a gene in the offspring
θ_{del}	deletion threshold
δ	for deciding when a classifier may be considered for deletion
θ_{sub}	subsumption threshold
$P_{\#}$	probability of using a # in a classifier gene when covering
P_I	initial value for the *prediction* parameter in a newly created classifier
ε_I	initial value for the *prediction error* parameter in a newly created classifier
F_I	initial value for the *fitness* parameter in a newly created classifier
P_{explr}	probability of random action selection
θ_{mna}	minimal number of actions that must be present in a match set [M] to avoid covering

Table 3.1: XCS parameters.

3.4.4 Relationship with Q-learning

In the description of XCS's Reinforcement component (section 3.4.2.2), the calculation of the payoff P and prediction p were given as:

$$P = r_{t-1} + \gamma \max_i P(a_i) \tag{3.2}$$

$$p \leftarrow p + \beta(P - p) \tag{3.3}$$

Substituting P in (3.3), the update of p becomes:

$$p \leftarrow p + \beta[r_{t-1} + \gamma \max_i P(a_i) - p] \tag{3.8}$$

which mirrors the update rule in Q-learning (section 2.4.3):

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

Thus, the update in XCS is directly related to that of Q-learning. In fact, XCS reduces to Q-learning when its generalisation mechanism is disabled. This is achieved by disabling the use of the wildcard symbol (#) and switching off the GA.[3] The effect of this is to produce a final classifier population that corresponds exactly to the Q-table entries for states that actually occurred during learning.

However, in this version of XCS, a classifier can represent only a single state-action pair (as in Q-learning). Therefore, the size of the XCS classifier population N must be large enough to represent every state that is observed during learning. For many sequential tasks this is not practical. Some form of generalisation over state *is* required. And this is where the main advantage of XCS lies. The wildcard symbol facilitates the representation of generalisations, while the GA facilitates the detection of generalisations. In the terminology of evolutionary algorithms, the generalisation mechanism of XCS facilitates the representation and detection of *categorical regularities* in the *payoff landscape*. In other words, regions of the state-action space that have the same expected reward are grouped together.

[3] In an XCS implementation this can be done by: (i) setting $P_{\#} = 0$ so that covering generates fully specific rules; (ii) setting θ_{GA} to a very large number so that the GA is never invoked.

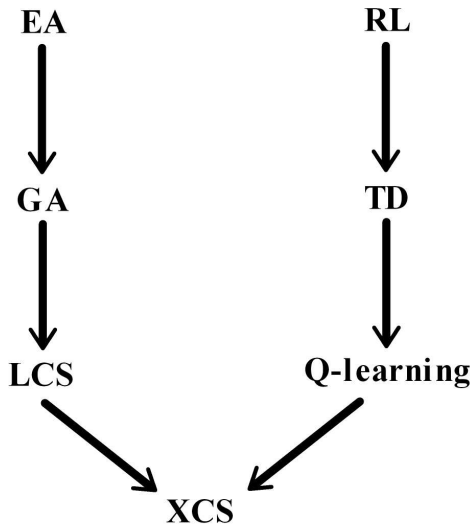

Figure 3.5: Conceptual origins of the evolutionary and reinforcement elements of the XCS algorithm.

To put the XCS algorithm in context, it is useful to consider the conceptual origins of the two elements of XCS (Figure 3.5). The evolutionary element is a modification of the one used in earlier Learning Classifier Systems (LCSs). The most important difference between XCS and LCS is that the fitness of rules is evaluated according to the accuracy of predicted rewards rather than by their magnitude. The transformation of rules is achieved using a Genetic Algorithm (GA), which, as I have indicated already (section 3.2), is a type of Evolutionary Algorithm (EA). The reinforcement element is directly related to the update rule used in Q-learning. This algorithm is a well known form of Temporal Difference (TD) learning, which, in turn, is one of several possible reinforcement learning (RL) algorithms.

In summary, XCS combines a genetic algorithm and a Q-learning like update rule to generate a population of condition-action rules that efficiently cover the state-action space. Indeed, XCS has been shown to be a proper generalisation of Q-learning (Wilson, 2000; Kovacs, 2002). With respect to sequential decision tasks, XCS is often described as an *evolutionary reinforcement learning* algorithm (Lanzi, 2002). Furthemore, XCS can produce much more compact mappings of the state-action space than is possible with tabular Q-learning. This allows much larger state-action spaces to be represented, which, in turn, allows more realistic sequential tasks to be solved. In the context of spoken dialogue systems, this implies that XCS might be applied to the learning of dialogue strategies in systems that require large state representations.

3.5 Learning dialogue strategies with XCS

The representation of individual state-action pairs in tabular RL algorithms creates a serious problem for developers of learned spoken dialogue strategies. The large state-action spaces that are likely to be required for realistic tasks cannot be represented in tabular form. However, the generalisation mechanism of the XCS algorithm does facilitate the representation of large state-action spaces. This suggests that an evolutionary approach to learning spoken dialogue strategies is worthy of investigation.

Therefore, I will develop an XCS-based framework for learning spoken dialogue strategies. This can be achieved using the same formalism employed by RL-based studies, namely, Markov Decision Processes (MDPs). Recall that a finite MDP is defined by: (i) a state set; (ii) an action set; (iii) a transition function; and (iv) a reward function. In an XCS implementation, state representations are encoded as binary strings. This is straightforward when representing boolean state variables. Other techniques are available for encoding integer and real values (e.g. Wilson, 2001). The action set in XCS can be represented as an ordered list. Reward functions can be encoded as lines of code within the XCS implementation. The framework will be implemented using Java JDK 1.5 (Sun, 2007).

In common with most approaches to learning dialogue strategies, the transition function will be modelled using a range of simulated users (sections 2.4.6 and 2.4.7.3). I intend to develop stochastic, goal-directed, hand-coded rule-based simulated users (Lin and Lee, 2001; Chung, 2004; Pietquin and Dutoit, 2006). A user model will consist of a set of probabilistic state-action rules. User behaviour will be conditioned on the current dialogue state (which will include a record of the last system utterance) and the user goal (e.g. fulfill a database query). I believe that this approach to implementing simulated users is sufficient to enable the XCS-based learner to experience enough user behaviours to generate robust human-computer interaction. Naturally, this hypothesis can only be properly tested through evaluations with human users.

3.6 Summary

Evolutionary Algorithms (EAs) are a group of stochastic search techniques inspired by Darwin's theory of evolution by natural selection. EAs operate on a population of potential solutions, often encoded in structures called *chromosomes*. This population of chromosomes is iteratively modified using operators borrowed from genetics. The idea is that successive generations of the population contain chromosomes that are better suited to their environment than the chromosomes from which they were created, just as in natural selection. In other words, chromosomes which encode the best solutions evolve over time and eventually dominate the population in a way that is similar to the notion of 'survival of the fittest'.

The XCS (eXtended Classifier System) algorithm makes extensive use of this idea of a population-based search for solutions. XCS combines an EA with a reinforcement learning (RL) mechanism to generate condition-action rules (*classifiers*) that can represent solutions for a variety of learning tasks. More specifically, XCS couples a genetic algorithm (GA) with a Q-learning update rule. With respect to sequential decision tasks, XCS is often described as an *evolutionary reinforcement learning* algorithm.

The principal advantage of XCS over temporal difference algorithms, such as Q-learning, is the ability to represent *regions* of the state-action spaces rather than single points (i.e. individual state-action pairs). This means that much larger state-action spaces can be represented, which in turn means that more realistic problems can be tackled. In the context of spoken dialogue systems, this implies that XCS might be applied to the learning of dialogue strategies in systems that require large state representations.

Chapter 4

Preliminary experiments

4.1 Overview

In this chapter, I explore how an implementation of the XCS algorithm can be used to develop simple dialogue strategies. I also investigate what effect varying the learning framework has on the resulting strategies. More specifically, I investigate how the state representation, reward function, simulated environment and learning parameters affect both the way in which a strategy is learned (e.g. how quickly) and what strategy is generated. If I can develop sensible dialogue strategies for this simple environment and reach an understanding of the effect of the learning framework on dialogue strategy learning, I can go on to develop strategies for much more realistic tasks, in particular those requiring very large state-action spaces.

4.2 Experimental aims

These preliminary experiments have two main aims. Firstly, to assess whether an implementation of the XCS algorithm can be applied to the development of dialogue strategies. Classifier systems have so far been applied to a small number of sequential tasks (e.g. Wilson, 1995; Smith et al., 2000). However, in these tasks the goal was relatively easy to define precisely. This is simply not the case with learning dialogue strategies; it is not clear what constitutes a 'win' in this domain. Therefore, it is

important to investigate whether XCS can generate dialogue strategies that would be considered sensible and useful by system developers.

Secondly, in the application of reinforcement learning to sequential decision tasks, the learned policy is directly related to a number of elements: (i) the state representation; (ii) the reward function; (iii) the environment; (iv) the learning parameters. The literature on the learning of dialogue strategies contains few empirical results describing the effect of these elements on the strategies that are learned (Frampton and Lemon, 2005; Pietquin and Dutoit, 2006; Reiser and Lemon, 2006). It is expected that this may also be the case with the application of XCS to learning dialogue strategies. Therefore, it is important to investigate experimentally the effect of these elements on the strategies generated by XCS.

4.3 Experimental methodology

In slot-filling dialogues, an effective strategy is one that interacts with the user in a satisfactory way while trying to minimise the length of the dialogue. A fundamental component of user satisfaction is the system's prevention and repair of any miscommunication between it and the user. These experiments were used to investigate whether XCS could learn simple slot-filling strategies, including the use of some essential miscommunication handling behaviours. I chose a simple hotel booking task. The goal of this experimental system was to acquire the values for three slots: the check-in date, the number of nights the user wishes to stay and the type of room required (single, twin etc.).

Both the XCS algorithm and the simulated users were implemented using Java JDK 1.5 (Sun, 2007). Unless specified otherwise, the XCS learning parameters (section 3.4.3) in all experiments were: $N = 1000$, $\beta = 0.2$, $\alpha = 0.1$, $\varepsilon_0 = 10$, $\nu = 5$, $\gamma = 0.95$, $\theta_{GA} = 50$, $\chi = 0.8$, $\mu = 0.04$, $\theta_{del} = 20$, $\delta = 0.1$, $\theta_{sub} = 20$, $P_{\#} = 0.33$, $p_I = 12$, $\varepsilon_I = 0$, $F_I = 0.01$, $p_{explr} = 0.5$. The value of θ_{mna} in each experiment was equal to the number of possible system actions. In each experiment, a dialogue strategy was allowed to evolve over a fixed number of dialogues (between 25,000 and 100,000) with a simulated user. Dialogues were limited to a maximum of 50 system dialogue acts. The total payoff (reward) for each dialogue was logged. Each experiment was repeated ten times (to produce an average result).

A moving average of the total payoff was plotted for each experiment to enable visual inspection of whether convergence towards an optimal strategy was achieved.

4.4 Experimental results

4.4.1 Experiment 1: A simple dialogue strategy

The goal of the first learned strategy is simply to greet the user, ask for the slot information, present the query results and then finish the dialogue, in that order. In all experiments, the presentation of the query results and closure of the dialogue were combined into a single dialogue act. Therefore, the dialogue acts available to the system for the first experiment were: *Greeting, Query+Goodbye, Ask(Date), Ask(Duration)* and *Ask(RoomType)*. Four boolean variables were used to represent the state of the dialogue: *GreetingFirst, DateFilled*, *DurationFilled*, *RoomFilled*.

It was important to experiment with the effect of reward functions at this early stage because a number of empirical results in the literature have noted that strategies generated by RL are sensitive to variations in the reward functions (Paek, 2006). It is important to assess whether this is the case with evolutionary reinforcement learning. Consequently, I looked at four aspects of the reward function: (i) absolute magnitude; (ii) relative magnitude; (iii) turn penalty; (iv) partial rewards.

4.4.1.1 Absolute Magnitude

In the first set of experiments, I examined the effect of varying the absolute magnitude of the rewards. I created six versions of the reward function. Each function maintained the same ratio between the reward for taking a system action and the reward for completing the goal (1:100), but differed in absolute terms. At the end of each dialogue, a penalty was assigned to each action performed and a larger reward was given if the goal of filling the three slots was achieved (Table 4.1). For example, the total possible payoff associated with the optimal strategy in Experiment 1a was: $-1 \times 5 + 100 = 95$, i.e. -1 for each of the five actions required to complete the dialogue and 100 for filling the three slots.

Exp.	Per Action	Goal Completion	Max Payoff
1a	-1	100	95
1b	-10	1,000	950
1c	-100	10,000	9,500
1d	-1000	100,000	95,000
1e	-10,000	1,000,000	950,000
1f	-100,000	10,000,000	9,500,000

Table 4.1: Reward functions for Experiments 1a–1f.

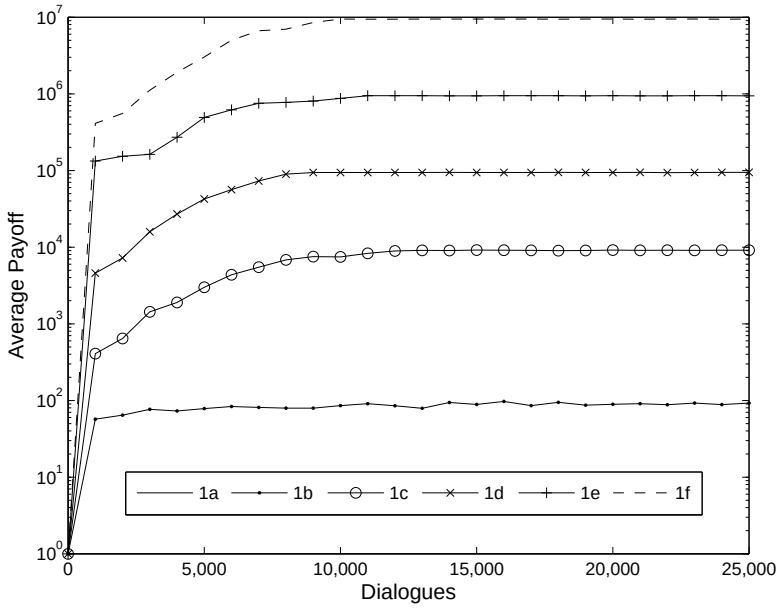

Figure 4.1: Strategy convergence: Experiments 1a–1f (absolute reward).

A plot of the average payoff for each experiment is given in Figure 4.1. It shows that an optimal strategy is learned in Experiments 1c–1f within approximately 10,000 dialogues. However, in Experiments 1a and 1b, the maximum payoff of 95 and 950 respectively, were reached after 10^6 dialogues (on average). The plotted value of Experiment 1a in Figure 4.1 is zero. Larger magnitude rewards appear to generate optimal behaviour much more quickly. I could not find any explanation for these observations either in the literature or through personal communication with members of the XCS community. Nevertheless, to take advantage of this observation, I

implemented a line in my code to multiply the value of all subsequent rewards by 1000. This allowed me to generate optimal strategies quickly using reward functions expressed in units that are understood more clearly.

4.4.1.2 Turn penalty

Before proceeding further, it was useful to verify that a turn penalty is actually needed. In many types of sequential decision tasks, such as maze learning (Wilson, 1995) and game playing (Tesauro, 1992), a small negative reward is assigned to each time step. This is used to encourage the learner to complete the task as quickly as possible. Previous work on the use of RL for dialogue strategy learning report some kind of reward measure like this (e.g. Frampton and Lemon, 2005; Cuayáhuitl et al., 2006a) This is frequently referred to as a *turn penalty*. Experiments 1a–1f were all repeated, this time without the turn penalty. That is, only the reward for goal completion was assigned. In all cases, results indicated that the learner settled on a sub-optimal strategy. That is, the learner completed the task but not in the shortest possible number of turns.

4.4.1.3 Relative Magnitude

Previous work has investigated the relative size in magnitude between interim and final rewards using Q-learning for generating dialogue strategies (English and Heeman, 2005). In this series of experiments, I examined whether varying the ratio between the turn penalty and the reward for goal completion would have any effect on the learned strategy. Four reward functions were devised (Table 4.2). The ratio of the turn penalty to the goal completion reward was incrementally increased from 1:100 to 7.5:100. A plot of average payoff for each experiment is given in Figure 4.2.

Exp.	Per Action	Goal Completion	Max Payoff
1g	1.0	100	95.0
1h	2.5	100	87.5
1i	5.0	100	75.0
1j	7.5	100	62.5

Table 4.2: Reward functions for Experiments 1g–1j.

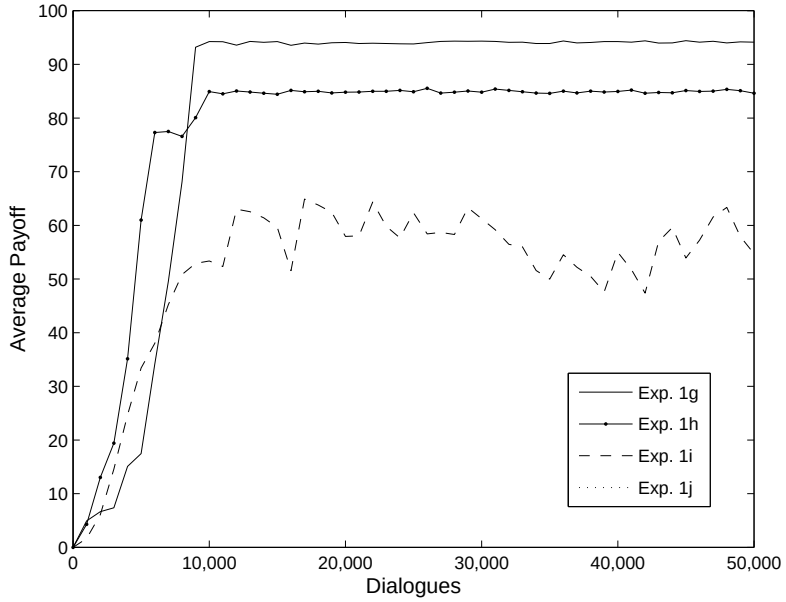

Figure 4.2: Strategy convergence: Experiments 1g–1j (relative reward).

Results showed that no useful strategy was generated when the ratio of turn penalty to goal completion was 7.5:100 (Experiment 1j). The plotted value of Experiment 1j in Figure 4.2 is zero. This experiment was allowed to run for 10^6 dialogues but showed no change. The strategy generated with a reward ratio of 5.0:100 appeared to oscillate between two sub-optimal strategies (Experiment 1i). Similar experiments were performed with higher ratios; all failed to generate optimal strategies. Optimal strategies were generated when the ratio was 1:100 (Experiment 1g) or 2.5:100 (Experiment 1h). These initial results suggest that a relatively high turn penalty ratio may produce sub-optimal strategies.

4.4.1.4 Partial Rewards

In this set of experiments, I examined the effect of partial rewards. The aim was to investigate whether dividing the goal completion reward into sub-rewards affected the rate of convergence towards the optimal strategy. Previous work has shown that applying partial rewards to tabular Q-learning can produce sub-optimal conversational strategies (Frampton and Lemon, 2005, section 4.3). In Experiments 1k–1m, the maximum possible payoff was the same. They differed only in the way the goal completion reward was structured (Table 4.3).

Exp.	Per Action	Goal Completion	Max Payoff
1k	-1	100 for greeting first and filling all 3 slots	95
1l	-1	50 for greeting first; 50 for filling all 3 slots	95
1m	-1	25 for greeting first; 25 for each filled slot	95

Table 4.3: Reward functions for Experiments 1k–1m.

In Experiment 1k, a reward of 100 was given if the first system act was a greeting and all three slots were filled. In Experiment 1l, the rewards for greeting first and slot-filling were sub-divided (50 points each). In Experiment 1m, the reward structure was further subdivided: 25 points were awarded equally for greeting first and filling a single slot. The plots for the strategies generated in these experiments are given in Figure 4.3. All three strategies were optimal. The experiment rewarding only the complete accomplishment of the goal (Experiment 1k) converges after approximately 10,000 learning dialogues. In Experiment 1l, where the greeting and slot-filling tasks were separated, an optimal strategy took much longer to learn, after approximately 40,000 dialogues. On the other hand, the completely decomposed reward model (Experiment 1m) converges towards the optimal strategy most rapidly, after approximately 6,000 dialogues.

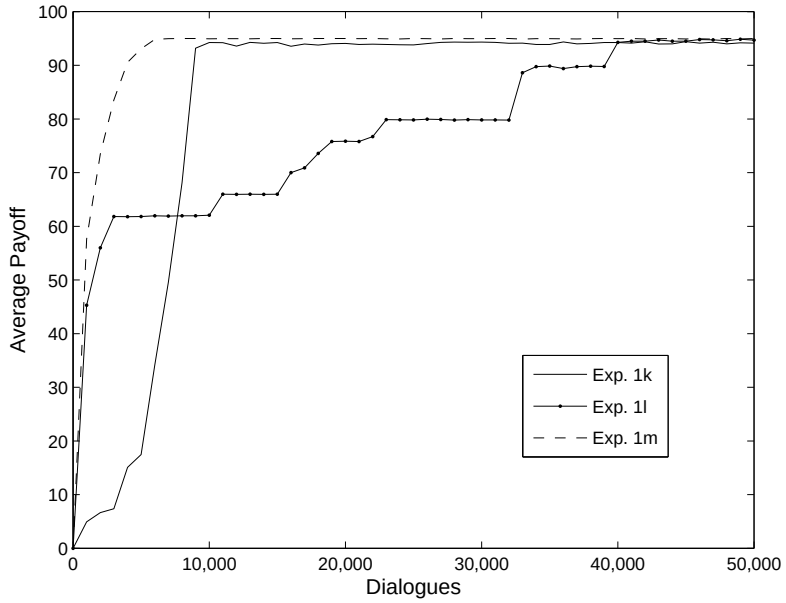

Figure 4.3: Strategy convergence: Experiments 1k–1m (partial reward).

These results differ slightly to those reported in Frampton and Lemon (2005), where partial rewards were investigated within a RL framework. In this study, the best reward function was the one where only the complete goal accomplishment was rewarded ("all-or-nothing reward"). However, since the experimental framework in that study differed from the one used in the current study, it is not possible to draw any firm conclusions as to whether partial rewards are better. Further research is required.

4.4.2 Experiment 2: Handling miscommunication

In this set of experiments, the functionality of learned strategies was extended through the addition of extra state variables and dialogue acts. More specifically, these experiments focus on learning essential behaviours for handling miscommunication arising from speech recognition errors. Experiments 2a and 2b added a new dialogue act: *Ask(All)*. The goal here was to ask for all three slot values if the probability of getting the slot values was reasonably high. If the probability was low, the system should ask for the slots one at a time as before. This simple characterisation of 'channel noise' was modelled in the simulated users by two variables: *Prob1SlotCorrect* and *Prob3SlotsCorrect*. The values for these variables in Experiments 2a and 2b respectively were: 0.9 and 0.729 ($=0.9^3$); 0.5 and 0.125 ($=0.5^3$).

Figure 4.4 shows the strategy convergence plots for the 25,000 training dialogues in Experiments 2a and 2b. It shows that the algorithm learned to ask for all three slots when the probability of success is high (*Prob3SlotsCorrect* = 0.729) and to ask the simulated user for individual slots when the recognition accuracy is low (*Prob3SlotsCorrect* = 0.125). The average payoff for Experiment 2b was less than 2a since it required more turns on average to successfully acquire the slot values.

Experiments 2c and 2d added three new dialogue acts: *Explicit_Confirm(Date)*, *Explicit_Confirm(Duration)*, *Explicit_Confirm(RoomType)* and three new state variables: *DateConfirmed*, *DurationConfirmed*, *RoomConfirmed*. The goal here was for the system to learn to confirm each of the slot values after the user has given them. Experiment 2d sought to reduce the dialogue length further by allowing the system to confirm one slot value while asking for another. Two new dialogue acts were available in this experiment: *Implicit_Confirm(Date)+Ask(Duration)* and *Implicit_Confirm(Duration)+Ask(RoomType)*.

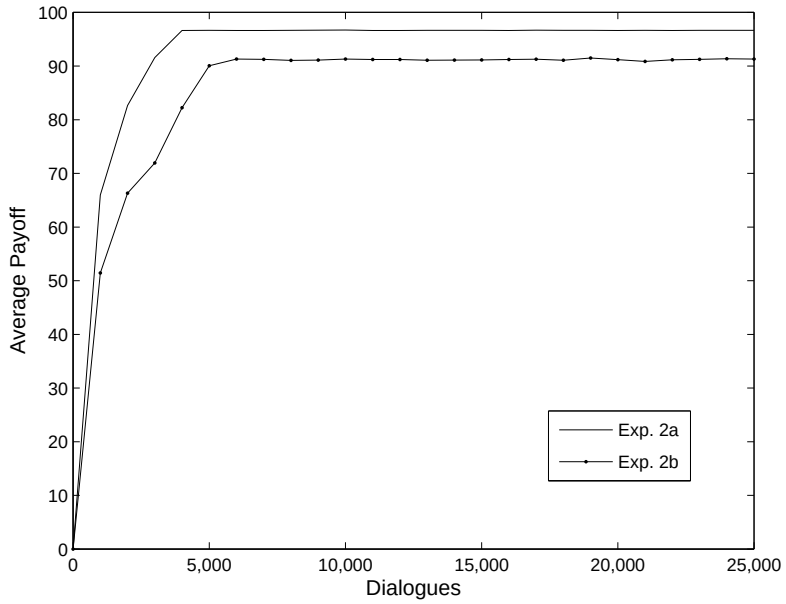

Figure 4.4: Strategy convergence: Experiments 2a–2b (channel noise).

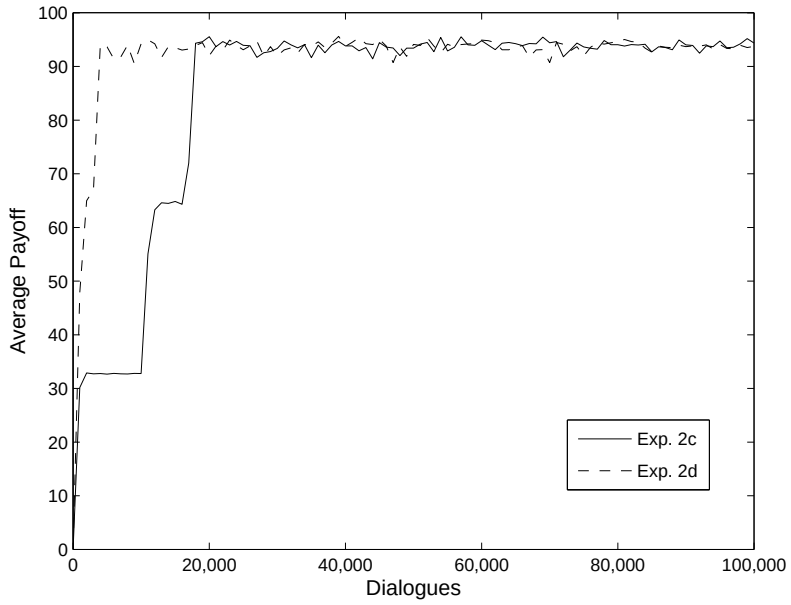

Figure 4.5: Strategy convergence: Experiments 2c–2d (confirmations).

Figure 4.5 shows the strategy convergence plots for the 100,000 training dialogues in Experiments 2c and 2d. It shows that Experiment 2c approached the optimal strategy (i.e. fill and confirm all slots in the shortest time) after approximately 20,000 dialogues whereas Experiment 2d converged after approximately 5000 dialogues. This is encouraging because it suggests that XCS remains focused on finding the shortest successful dialogue even when the number of available actions increases. In other words, it has learned to make appropriate use of the implicit confirmations.

4.4.3 Experiment 3: User modelling

Previous work has shown that the quality of a learned strategy can be dramatically affected by the quality of the simulated user with which it interacts during learning (Ai and Litman, 2006; Schatzmann et al., 2006). Therefore, it was important at this early stage in the use of XCS for dialogue strategy learning to investigate the effect of different simulated users on the quality of learned strategies. To do this, three different types of simulated user were defined. User A was designed to act as a less cooperative user. It did not respond to the system's requests for slot values or confirmation prompts 20% of the time. User B was the same as the one used in Experiment 2c. That is, it was a cooperative user, responding to the system's prompts in a cooperative manner (but was not over-informative). System C was modelled as a highly cooperative or expert user; 50% of the time this user was over-informative, giving more slot values than were asked.

The strategy convergence plots for Experiments 3a–3c are given in Figure 4.6. It is interesting to note that the less cooperative user (Experiment 3a) took significantly longer to converge on the same optimal strategy. Similarly, learning with the highly cooperative/expert user (Experiment 3c) converged most rapidly but not significantly more than User B (Experiment 3b). The performance of the three learned strategies were compared using a three-fold cross validation procedure. That is, the strategy learned using each simulated user was then tested against 10,000 simulated test dialogues with all three simulated users (Table 4.4). The results showed that the learned strategies all performed optimally with each of the three users. Of course, the strategies that were learned were still relatively simple. Nevertheless, the results so far suggest that the dialogue strategy learning method is robust with respect to a variety of simulated behaviours.

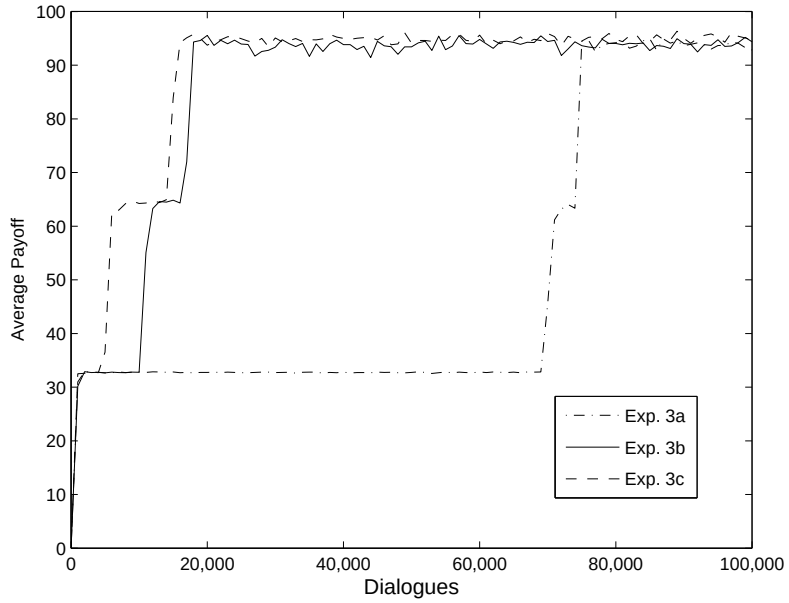

Figure 4.6: Strategy convergence: Experiments 3a–3c (user modelling).

Sim. user (training)	Sim. user (testing)	Average Payoff
A	A	94.06
A	B	95.23
A	C	96.79
B	A	94.24
B	B	94.89
B	C	96.45
C	A	94.14
C	B	95.28
C	C	96.96

Table 4.4: Cross validation results for simulated users in Experiments 3a–3c.

4.4.4 Experiment 4: Incorporating domain knowledge

Several studies on generating dialogue strategies have used RL to enhance existing strategies. In these studies, RL has been used to explore parts of the policy (strategy) space that have not been explored by supervised learning (Henderson et al., 2005) or by the initial hand-coded strategy (Litman et al., 2000; Walker, 2000). There is a good practical argument for doing this - why waste time learning part of a strategy that you (as a developer) already know should happen? For example, it is obvious that the system should greet the user at the very beginning of the dialogue and not at any other time.

4.4.4.1 State representation

In all the previous experiments, a boolean variable *GreetingFirst* was used to indicate whether the first system dialogue act was a greeting. Since it is obvious that a dialogue should always begin with a greeting, it seems sensible to assume that this situation has occurred and then learn a strategy from this point onwards. Since the learned strategy is 'smaller' as a result, it seems intuitive that it will take less time to learn the reduced strategy. In fact, the removal of a boolean state variable halves the size of the state space.

Experiment 4a is the same as Experiment 2c. Experiment 4b is the same as 4a, but with the *GreetingFirst* variable removed from the state space and the dialogue act *GreetingFirst* removed from the list of possible system acts. The plot of strategy convergence (Figure 4.7) demonstrates that the reduced strategy was indeed learned much more quickly. It has a slightly higher average payoff since it takes one turn less to complete (i.e. there was no greeting).

In a practical implementation, a complete strategy would have to combine the hard-coded rules (e.g. for greeting) and the learned rules. It is straightforward to imagine a procedure for doing this. In the execution of the hybrid strategy the dialogue manager could compare the current dialogue state with the list of conditions in the hard-coded rule set. If any conditions matched, the appropriate rule(s) would be executed. If there are no matches, the dialogue manager could then search the conditions covered by the learned strategy.

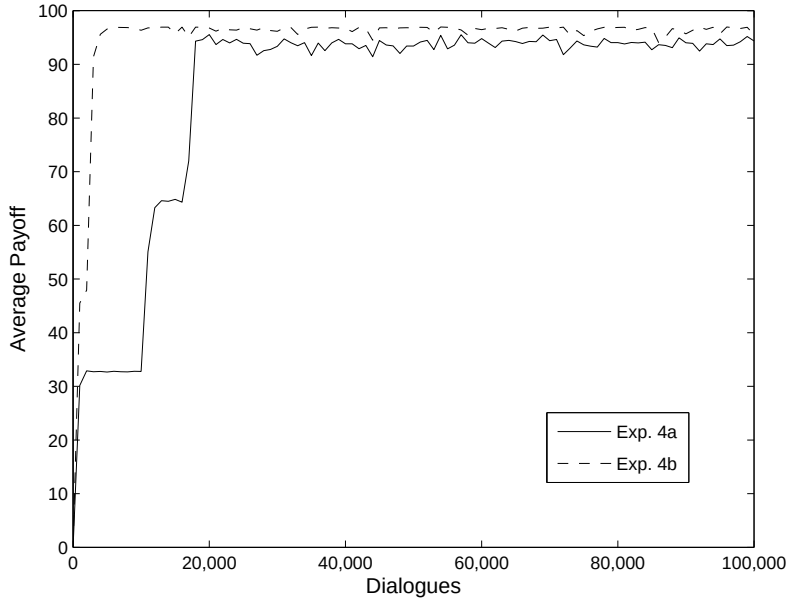

Figure 4.7: Strategy convergence: Experiments 4a–4b (hard-coded greeting).

4.4.4.2 Slot Ordering

In a hotel booking task, there is probably no standard order in which slot values are requested. It is entirely possible that a hotelier could ask for any piece of information first (e.g. check-in date, duration, room type). In other domains, however, it is more natural to ask the user for some pieces of information first. For example, in a travel information or booking system, it would seem very unnatural to ask a user for a preferred return date before asking for the date of the outward journey. Of course, the user should be able to take the initiative and answer the questions in any order s/he wishes. Nevertheless, in order to maximise user satisfaction, it makes sense to encode domain knowledge of this nature.

Inspection of the logs of the learned strategies showed that the three slots were requested in random order. This is not surprising since there was nothing in the learning framework (state representation, reward function etc.) that would encourage the learner to ask for slot information in a certain order. Therefore, in applications where slot ordering is important, the learner needs some indication that this is part of the overall goal. Otherwise, realising such a goal would simply be wishful thinking on the part of the developer.

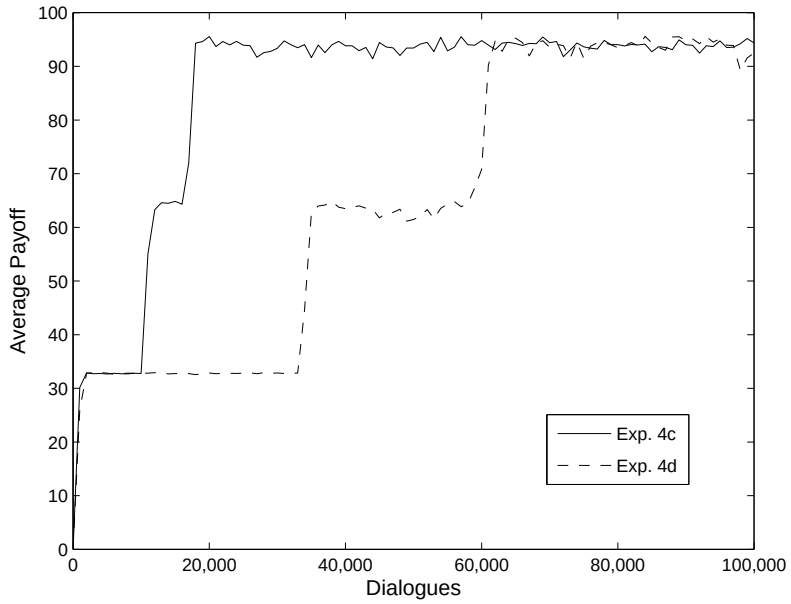

Figure 4.8: Strategy convergence: Experiments 4c–4d (slot ordering).

The requirement to ask for slot values in a specific order can be encoded in a number of ways. For example, the ordering could be incorporated into an initial hand-coded strategy that is later optimised by learning. Similarly, the slot ordering could be included in an initial strategy created using a corpus and supervised learning methods (cf. Henderson et al., 2005). Another approach is to simply have a dialogue act that asks for the next unfilled slot in an ordered list (Cuayáhuitl et al., 2006b).

Alternatively, we can encode this domain knowledge in the reward function. That is, the developer can insert a rule into the reward function that generates a large penalty if something undesirable occurs. Within the current framework, an example could be represented informally as: "*if slot* ***z*** *is asked for before slot* ***x*** *or* ***y*** *then the current reward = -100,000*". This approach makes it easy for a developer to hand-code events that should always (or never) happen. It is important that a developer retains control over the behaviour of a learned strategy.

Experiments 4c–4d illustrate this idea. Experiment 4c is the same as 2c. Experiment 4d sets up the order of slots to be filled as *Date*, *Duration* and *RoomType*. To implement this slot ordering, the reward function assigned an immediate negative reward if a slot was filled or confirmed before an 'earlier' one. For example, the act of asking the user for the *Duration* slot value before the *Date* was filled or confirmed was assigned a reward of -100,000. In the plot of the strategy convergence (Figure 4.8) it can be seen that implementing this ordering effect did take longer to learn.

4.4.5 Other experiments

A number of additional experiments were conducted in order to examine other approaches to improving the performance of the strategy learner. These approaches included: (i) varying XCS learning parameters; (ii) investigating the effect of positional semantics; and (iii) implementing an automatic termination criteria. It is not necessary to present all of the results in graphical or tabular form. In the following sections I will only summarise the results for each of these approaches.

4.4.5.1 XCS learning parameters

The values for the XCS learning parameters used in the previous experiments were given in section 4.3. General guidelines for selecting these values are given in Butz and Wilson (2002). It was instructive to vary the value of some of these parameters to see what effect, if any, they had on the learning of dialogue strategies. More specifically, I experimented with: the number of microclassifiers (N), the learning rate (β); the discount factor (γ), the crossover rate (χ) and the mutation rate (μ). In some experiments, results were slightly improved; in other cases performance was drastically diminished. However, no consistent improvements were achieved. Personal communication with members of the XCS community confirmed that few of the parameters were ever changed. For example, Lanzi (2006) reported that "practically speaking, many of the parameters could be regarded as system constants" .

4.4.5.2 Positional semantics

In the context of learning classifier systems, positional semantics refer to how the same state information can be represented in different ways in the condition part of a classifier rule (Schuurmans and Schaeffer, 1989). For example, in Experiment 2c, the condition part of an XCS rule consisted of a seven bit string representing the seven boolean variables: *GreetingFirst*, *DateFilled*, *DurationFilled*, *RoomFilled*, *DateConfirmed*, *DurationConfirmed*, *RoomConfirmed*. For example, the string "1100100" meant that the user was greeted at the start of the dialogue and that the *Date* slot had been filled and confirmed.

A simple example of altering the positional semantics is to set the condition string to represent the boolean variables in a different order (e.g. *GreetingFirst*, *DateFilled*, *DateConfirmed*, *DurationFilled*, *DurationConfirmed*, *RoomFilled*, *RoomConfirmed*). Several experiments were conducted with alternative arrangements for the state representation. No differences in the generated strategies were detected. Discussions with members of the XCS community confirmed that no-one had observed issues with alternative positional representations.

4.4.5.3 Termination criteria

The use of automatic termination criteria has already been identified as one of the challenges for implementing a reliable dialogue strategy learner (see section 2.4.7). For XCS, one mechanism for auto-termination has been outlined (Kovacs, 1997). However, it is suggested that this mechanism is successful only if the environment is deterministic, which is not the case with dialogue systems. No other methods for automatic termination have been found in the XCS literature.

I experimented with a number of alternative termination criteria. In one experiment, the algorithm was terminated when X% (e.g. 95%) of the last Y (e.g. 500) training dialogues contained the same strategy. In another experiment, the algorithm was terminated when 95% of the last Y (e.g. 1000) training dialogues had approximately the same total payoff. A number of XCS parameters were also investigated as possible candidates for detecting convergence (e.g. system error, fitness etc.). However, none of these approaches produced results that were consistent enough to be used reliably. If we are to build a software tool for learning dialogue strategies, it is important that the strategy learner terminates automatically when an optimal solution has been generated. This is an area for future research. However, for the purpose of developing dialogue strategies experimentally, visual inspection of strategy convergence plots are sufficient.

4.5 Discussion

These preliminary experiments demonstrated that XCS can be used to develop dialogue strategies that are optimal with respect to a reward function. This learning framework was used to develop a slot-filling strategy for a simple hotel booking task. The

generated strategies exhibited a number of behaviours that are essential for a good dialogue strategy. For instance, the learned dialogue strategies took account of simple models of error recognition to adapt its behaviour accordingly (e.g. to ask for several slot values if recognition accuracy is high, otherwise to ask for slot values one at a time). The strategies also made use of explicit and implicit confirmation prompts. It should be noted, however, that it was straightforward to verify whether a learned strategy was optimal with respect to its reward function. In more complex, stochastic environments (i.e. simulated users) assessing the 'optimality' of a learned strategy is much more difficult.

The experiments also demonstrated how domain knowledge can be incorporated into a learned strategy. This can serve two purposes. Firstly, to produce a behaviour that might not otherwise occur. The requirement that slots be filled in a certain order is a very good example (section 4.4.4.2). Although this addition increased the time taken to learn the optimal strategy, such a requirement might be considered essential for certain tasks. Therefore, it is important to be able to facilitate such a requirement. By contrast, domain knowledge can also be used to accelerate learning, dramatically in some cases. For example, by removing the subgoal of greeting the user at the start of a dialogue (an obvious thing to do), the dialogue strategy in section 4.4.4.1 was generated much more quickly, requiring approximately 2,000 training dialogues instead of 20,000.

4.6 Summary

The XCS algorithm can be employed to develop useful dialogue strategies for an information-seeking (slot-filling) task. In these preliminary experiments, the learned strategies exhibited a number of essential behaviours in conjunction with a variety of simple simulated user types and models of recognition error. These preliminary experiments have helped to answer the questions of what can be achieved with XCS and how. However, the size of the state-action space is very small. The largest space contained 1024 unique state-action pairs. What is required now is to scale up to a more realistic problem requiring a much larger state-action space.

Chapter 5

Learning spoken dialogue strategies with large state-action spaces

5.1 Overview

In this chapter, I focus on the learning of dialogue strategies where the state-action spaces are very large. These solution spaces cannot be represented using traditional reinforcement learning (RL) algorithms, such as Q-learning; the space and time complexity of these algorithms with large problems are prohibitive. In the context of RL problems, "large" means a state-action space that would require greater than, say, one million (10^6) elements when represented in tabular form. In realistic spoken dialogue applications, state-action spaces can very easily reach or exceed this figure.

I examine whether XCS can learn useful dialogue strategies by extending the hotel booking task in the previous chapter. The strategy facilitates a greater range of system and user acts and a richer state representation. The size of the state-action space is 4.6×10^9, much too large to represent in tabular form. I also examine the effect of reward functions. In particular, I investigate how much detail is required in the reward function for the XCS algorithm to learn the developer's intended goal. Finally, I investigate the effect of different simulated environments on the learning of dialogue strategies. I assess whether probabilistic, goal-directed hand-coded simulated users continue to be sufficient for the development of useful dialogue strategies.

5.2 Experimental aims

These experiments have three main aims. Firstly, I need to assess whether an implementation of the XCS algorithm can be applied to the development of dialogue strategies that require large state-action spaces. Secondly, I wish to investigate further the effect of reward functions on the behaviour of learned strategies (e.g. whether they reflect the developer's intended goal). Thirdly, I want to investigate further whether hand-coded simulated users are still sufficient for learning more sophisticated dialogue strategies.

5.3 Experimental methodology

The hotel booking task presented in the previous chapter is extended in a number of ways. First, a fourth slot value is added to the original three (check-in date, number of nights, room type), namely *location*. In other words, the user should be able to specify which location (e.g. town) s/he wishes to stay. Second, the number of system and user dialogue acts is extended to represent a more realistic interaction (see section 5.4.1). For example, a user can ask for system utterances to be repeated, or can ask the system for context-sensitive help. Therefore, the goal of the experimental system remains roughly the same: to acquire values for slots while trying to minimise the length of the dialogue. However, the system will be required to handle more complex dialogues, resulting from a richer state representation and greater number of system and user dialogue acts.

The dialogue strategies continued to be developed in conjunction with stochastic, goal-directed, hand-coded simulated users. These simulations are represented by a set of probabilistic state-action rules (Chung, 2004; Pietquin and Dutoit, 2006). An example is given in section 5.4.3. User behaviour is conditioned on the current dialogue state (which includes a record of the last system and user utterances) and the user goal (e.g. fulfill a database query). I believe that this approach to implementing simulated users is sufficient to enable the XCS-based learner to experience enough variety of user behaviour to generate robust human-computer interaction. Naturally, this hypothesis can only be properly tested by evaluations with human users (see Chapter 6).

Both the XCS algorithm and the simulated users were implemented using Java JDK 1.5 (Sun, 2007). Unless specified otherwise, the XCS learning parameters (section 3.4.3) in all experiments were: $N = 1000$, $\beta = 0.2$, $\alpha = 0.1$, $\varepsilon_0 = 10$, $\nu = 5$, $\gamma = 0.95$, $\theta_{GA} = 50$, $\chi = 0.8$, $\mu = 0.04$, $\theta_{del} = 20$, $\delta = 0.1$, $\theta_{sub} = 20$, $P_{\#} = 0.33$, $p_I = 12$, $\varepsilon_I = 0$, $F_I = 0.01$, $p_{explr} = 0.5$, $\theta_{mna} = 15$ (i.e. the number of possible system actions).

In each experiment, a dialogue strategy was allowed to evolve over a fixed number of dialogues (between 200,000 and 500,000) with a simulated user. Dialogues were limited to a maximum of 50 system dialogue acts. The total payoff (reward) for each dialogue was logged. Each experiment was repeated ten times (to produce an average result). A moving average of the total payoff was plotted for each experiment to enable visual inspection of whether convergence towards a strategy was achieved. In the experiments that follow, it is more difficult to assess what constitutes an optimal dialogue strategy. This is because of the larger state representation, larger system and user action sets and the more sophisticated, stochastic simulated environment. Instead, we can use the convergence plot to detect when strategy convergence occurs and then inspect the logged training dialogues.

5.4 Experimental results

5.4.1 Experiment 5: A large state-action space

The chosen system and user dialogue acts are summarised in Table 5.1. The system dialogue acts allow the system to ask the user for the slot values or to confirm these values, either explicitly or implicitly (i.e. while asking for another slot value). The system can restart or end the dialogue, give the user help or hand the user on to a 'human operator'. The system can then present the results of a user's database query. The user dialogue acts allow the user to terminate the dialogue, ask for help, ask the system to repeat its last utterance, start the dialogue from the beginning or provide slot information. The user can also confirm or reject a system's explicit confirmation of slot values. The user can provide the system with more than one slot value in a single utterance, including slot values that were not asked for by the system in order to reduce the length of the dialogue. The *unknown* act represents a user utterance that cannot be interpreted satisfactorily.

System acts	User acts
GREETING	command(bye)
GOODBYE	command(request_help)
HANDOFF	command(request_repetition)
GIVE_HELP	command(restart)
RESTART	answer(yes)
REQUEST_INFO(check-in)	answer(no)
REQUEST_INFO(nights)	provide_info(check-in) *
REQUEST_INFO(roomType)	provide_info(nights) *
REQUEST_INFO(location)	provide_info(roomType) *
REQUEST_INFO_WITH_IMPCONF(check-in)	provide_info(location) *
REQUEST_INFO_WITH_IMPCONF(nights)	unknown
REQUEST_INFO_WITH_IMPCONF(roomType)	
REQUEST_INFO_WITH_IMPCONF(location)	
EXPCONF(lo+med_confs)	* *multiple slot values can be*
DBASE_RESULTS	*provided in a single utterance*

Table 5.1: System and user dialogue acts for Experiment 5.

State variable	Possible values	Cumulative state space size
check-in_confidence	empty, lo, med, hi	4
nights_confidence	empty, lo, med, hi	16
roomType_confidence	empty, lo, med, hi	64
location_confidence	empty, lo, med, hi	256
check-in_times_asked	1..10	2,560
nights_times_asked	1..10	25,600
roomType_times_asked	1..10	256,000
location_times_asked	1..10	2,560,000
last_user_act	11 values (see Table 5.1)	28,160,000
last_system_act	15 values (see Table 5.1)	422,400,000

Table 5.2: State representation for Experiment 5.

The state variables used to represent the dialogue state were based on literature relevant to developing information-seeking tasks (e.g. McTear, 2004) and studies of dialogue strategy learning (Henderson et al., 2005; Pietquin and Dutoit, 2006; Frampton and Lemon, 2006). In common with many information-seeking tasks, the recognition confidences were represented by one of four values: 'empty', 'lo', 'med', 'hi'. A record of the number of times each slot was asked for or confirmed by the system was necessary to indicate that a dialogue was not progressing sufficiently, perhaps due to the persistent (simulated) misrecognition of a particular slot. In such a situation, the system should pass the user on to a human operator. The list of state variables and their possible values are given in Table 5.2. The third column in this table serves to illustrate how the size of the state space can grow rapidly. With 422,400,000 unique states and 15 system actions, this means that the size of the state-action space was 6.3×10^9.

The reward function was defined as: (i) a value of 100 is assigned if all four slots have a confidence value of 'hi' and a database query is actioned ; (ii) a turn penalty (-1) is assigned to each turn taken by the system. In the simulated environment, there are two ways for a slot to have a high confidence level: (i) it is given a 'hi' confidence when the value is provided by the user; or (ii) the simulated user responds positively to a system confirmation prompt.

In spite of the large state-action space, a strategy, which appears optimal with respect to the reward function, is learned within approximately 60,000 training dialogues (Figure 5.1). Inspection of the logs of the training dialogues show that the system does indeed learn to fill slots with high confidence, using confirmation prompts where necessary. Dialogues are terminated with a call to the database query action (DBASE_RESULTS). However, the logs also show that the system does not achieve all the desirable behaviour, given the range of available system and user dialogue acts. For example, it never hands off to a human operator, nor does it give help when requested to do so. These kinds of responses would lead to a low level of user satisfaction in a real system.

These strategy deficiencies are not surprising, since we are "programming by reward". The reward function describes the intended goal, albeit at a very high level. In other words, the learner will not acquire behaviour that it is not rewarded to do so. Therefore, a reward function should indicate that it is important to responding appropriately to requests for help, or hand off to a human operator when the dialogue is proving problematic. This result highlights an important aspect of learning dialogue strategies.

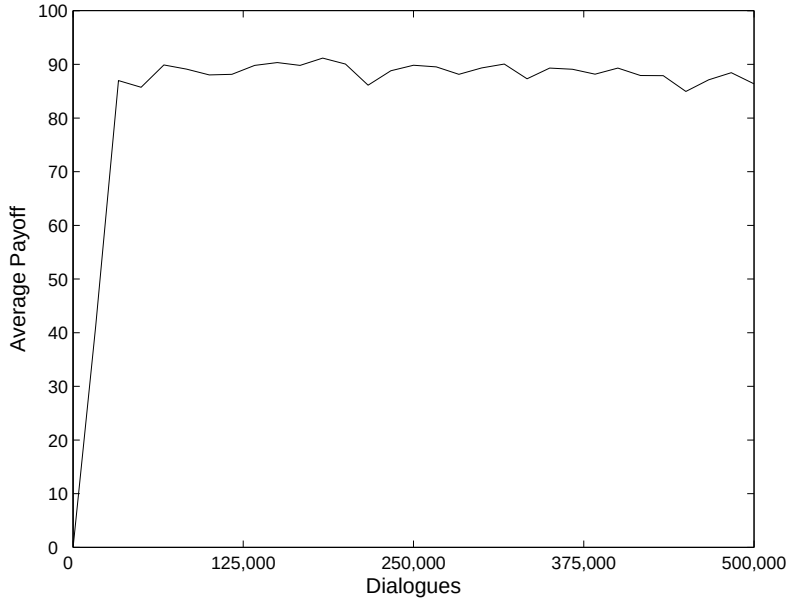

Figure 5.1: Strategy convergence: Experiment 5 (large state-action space).

5.4.2 Experiment 6: Reward functions

In this experiment, the reward function is extended to address the issues raised in Experiment 5. That is, it is crucial that the reward function accurately reflects the intended goal. Thus far, there is nothing in the reward function to motivate it to hand off to a 'human operator' when things are not going well. Similarly, a repeated request for help from the simulated user appears insufficient to motivate the system to provide help when required. Therefore, some model of punishment for ignoring user requests is required. The extended reward function is presented in Table 5.3. This type of reward function has been used in similar studies (e.g. Frampton and Lemon, 2005).

100	for filling all four slots with high confidence
50	for handing off to a human operator
-50	for each ignored user request for help, repetition, or restart
-1	for each turn taken

Table 5.3: Extended reward function for Experiment 6.

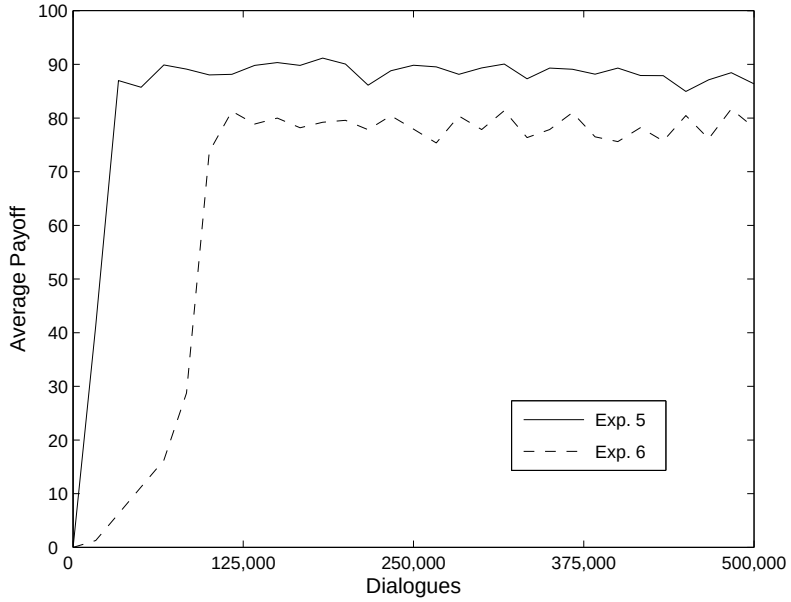

Figure 5.2: Strategy convergence: Experiment 6 (extended reward function).

The convergence plot for this new reward function shows that the learner takes longer to converge on a strategy (Figure 5.2). It seems likely that it has to learn a more sophisticated strategy and this takes more time. Inspection of the training logs show that the system now responds immediately to requests for help and occasionally hands off to the human operator when the simulated channel noise makes it difficult to fill some slots with high confidence. Also, the final average payoff is less than that in Experiment 5 since handing off results in a terminal reward of 50 (rather than 100 for filling the four slots).

5.4.3 Experiment 7: Simulated users

I have argued that a hand-coded, probabilistic, rule-based simulated user can generate sufficiently varied behaviour for the XCS algorithm to learn effective dialogue strategies. I have experimented with a number of alternative rules and representations. A pseudocode description of a typical user model is given in Figure 5.3. The variables, *LUA*, *LSA* and *CUA* refer to the last user act, the last system act and the current user act respectively. The purpose of the simulated user is to fill slots and generate the CUA. The variation in user behaviour is controlled by the probabilities associated with the rules. This simple framework can generate a wide variety of simulated user behaviour.

```
main
   if LUA = 'command(bye)' and is not done then CUA = 'command(bye)'
   if LUA = 'command(help)' and is not done then CUA = 'command(help)'
   if LUA = 'command(help)' and is done then fill a slot
   if LUA = 'command(restart)' and is not done then CUA = 'command(restart)'
   if LUA = 'command(restart)' and is done then reset state variables
   if LUA = 'command(repetition)' and is not done then LUA = 'command(repetition)'
   if LUA = 'command(repetition)' and is done then set LSA to previous LSA

   if LSA = 'EXPCONF' and slot X = 'lo' then
      set slot X = 'hi' with probability P = 0.50, CUA = answer(yes)
      else CUA = answer(no)
   if LSA = 'EXPCONF' and slot X = 'med' then
      set slot X = 'hi' with probability P = 0.70, CUA = answer(yes)
      else CUA = answer(no)

   if LSA = 'REQUEST_WITH_IMPCONF(X)'
      fill slot X with probabilities:  'lo' [P=0.20], 'med' [P=0.30], 'hi' [P=0.40]
      fill confirmable slots with probabilities:  'med' [P=0.30], 'hi' [P=0.50]

   if LSA = 'REQUEST_INFO(X)' then
      fill slot X with probabilities:  'lo' [P=0.20], 'med' [P=0.40], 'hi' [P=0.30]

   CUA = 'command(bye)' with probability P = 0.02
   CUA = 'command(help)' with probability P = 0.04
   CUA = 'command(restart)' with probability P = 0.03
   CUA = 'command(repetition)' with probability P = 0.07
   fill a slot with probability P = 0.80
   else CUA = 'unknown'


fill a slot
   X = randomly selected empty slot
   CUA = provide_info (X) with correct value with probability P = 0.8
      assign confidence of selected slot to:
         'lo'  [P=0.25]
         'med' [P=0.25]
         'hi'  [P=0.50]
   else CUA = provide_info (X) with incorrect value
      assign confidence of selected slot to:
         'lo'  [P=0.50]
         'med' [P=0.35]
         'hi'  [P=0.15]
```

Figure 5.3: Pseudocode example of a simulated user.

The purpose of this experiment was to examine the effect of learning a dialogue strategy in one simulated environment and executing the learned strategy in another environment. This is important because we need to assess whether dialogue strategies developed by XCS are highly dependent on the simulated environment. If this is the case, it implies that a dialogue strategy learned in conjunction with a less than realistic simulated environment may perform very poorly with real users (Schatzmann et al., 2006).

For the purpose of comparison, three different types of simulated user were defined. User A was designed to act as a less cooperative user; user B was more cooperative and user C was highly cooperative. Dialogue strategies were learned in conjunction with each of these users. The performance of the three learned strategies were then compared using a three-fold cross validation procedure. That is, the strategy learned using each simulated user was tested against 10,000 simulated test dialogues with all three simulated users (Table 5.4).

Sim. user (training)	Sim. user (testing)	Average Payoff
A	A	79.85
A	B	80.90
A	C	83.34
B	A	80.31
B	B	82.00
B	C	83.24
C	A	79.98
C	B	81.28
C	C	83.01

Table 5.4: Cross validation results for simulated users in Experiment 7.

The average payoff shown in these cross-validation results show that there is little difference between the three users. Test dialogues with User C have slightly better payoff because it has a higher probability of filling slots. Similarly, test dialogues with User A have slightly lower payoff because it is less cooperative. Inspection of the training logs indicate that the learned strategies execute the expected behaviour in a

variety of conditions. For example, where slot values are filled with low or medium confidence, they are confirmed by the system. The system also provides help or repeats a question when requested to do so by the user. When all four slots are filled with high confidence, the system ends the dialogue with a call to the database. When slots prove too difficult to fill, the system passes the user to a 'human operator'. It seem reasonable to conclude that the learned strategies are effective.

5.5 Discussion

Large state-action spaces are an important requirement for learning dialogue strategies in realistic tasks (Henderson et al., 2005). However, large state-action spaces cannot be represented using tabular RL methods, such as Q-learning. These experiments show that the XCS algorithm *can* develop dialogue strategies that employ a large state-action space. This is an important result – large state-action spaces can be tackled effectively using XCS.

The dialogue strategies developed in these experiments were optimal with respect to pre-defined reward functions. This is what was intended. In fact, this is the point of learning in sequential decision making – to "program by reward". However, it was also shown that the reward function must reflect the intended goal, albeit at a very high level. Otherwise, the behaviour required by the developer will never be learned. However, it is not always obvious what should be included in the reward function and what should be left to the learning algorithm to discover. In the context of spoken dialogue systems, further work will be required to understand more fully, the level of detail that is required in the reward function.

The use of stochastic, hand-coded simulated users appears to be sufficient for the development of sensible dialogue strategies. It could also be argued that hand-coded, rule-based simulated users make it relatively easy to test the performance of a learned dialogue strategy in particular circumstances. For example, it might be desirable to assess how a dialogue system copes with certain extreme user behaviours such a very low level of user cooperativity. It is less obvious how this might be achieved with a data-driven user simulation.

Finally, it should be noted that the learned dialogue strategies all appear to behave in a sensible manner. In fact, it appears that the dialogue strategies developed here would compare favourably with those developed by an experienced system developer using the traditional test-and-refine method (section 2.3). However, this is an informal observation. To assess the performance of a learned dialogue strategy more rigorously, an evaluation with human users is necessary.

5.6 Summary

XCS can learn dialogue strategies where large state-action spaces are required. This is an important advantage of this evolutionary algorithm over traditional reinforcement learning algorithms, such as Q-learning. In realistic spoken dialogue applications, large state-action spaces are often required. In an extension of the hotel booking task presented in the previous chapter, a state-action space with 6.3×10^9 unique state-action pairs, was represented. In spite of this, effective dialogue strategies were learned by XCS. It was also shown that reward functions must encapsulate all the required behaviour, albeit at a high level of abstraction. Finally, the use of hand-coded simulated users appears to be sufficient for the development of sensible dialogue strategies. However, to assess the performance of a learned dialogue strategy more accurately, an evaluation with human users is required. This is the goal of the next chapter.

Chapter 6

Evaluation

In the previous chapter, I demonstrated that the XCS algorithm and a hand-coded simulated user can be employed to learn dialogue strategies that require large state-action spaces. This is an important result for two reasons.

Firstly, practical dialogue strategies almost inevitably require state representations that are too large to be represented in tabular form. This means that traditional reinforcement learning algorithms cannot be applied. Some form of state generalisation is required. A key feature of the XCS algorithm is its ability to generalise quickly over large state spaces. Secondly, it was possible to learn useful strategies without the use of any pre-existing corpus of dialogues. The stochastic, goal-directed models of user behaviour were sufficient. Clearly, hand-coding a simulated user is much less time-consuming than collecting a corpus and using supervised learning algorithms to induce a model of user behaviour.

However, two questions must be addressed. First, can this approach produce a dialogue strategy that would perform well in a working spoken dialogue system, with real users? Second, would such a learned strategy perform as well as one devised by an experienced system developer? A framework for generating dialogue strategies quickly is of little use if the resulting performance is evaluated poorly by real users, or is inferior to hand-coded strategies. Therefore, I conducted an evaluation with human users of a dialogue strategy developed by XCS. I compared its performance with a baseline hand-coded strategy. Both strategies controlled a flight booking system using a live database.

In this chapter, I begin with a summary of previous work on evaluating learned dialogue strategies and highlight the important differences between this work and the present study. I outline my experimental aims and present a methodology based on the flight booking domain. I present the evaluation results and a discussion of their implications.

6.1 Related work

To date, three studies have evaluated learned strategies with real users. In both the ELVIS (Walker, 2000) and NJFun (Litman et al., 2000) systems, elements of a hand-coded strategy were improved by applying reinforcement learning. In ELVIS, a speech-based email reading system, evaluations of the optimised strategy indicated that users preferred the use of: (i) system initiative (rather than mixed initiative); (ii) summarisation of emails by either subject or sender, depending on the context, but not both; (iii) automatic reading of the first email in a group without prompting by the user to do so. The NJFun system provided users with information about things to do in New Jersey. The system asked users for a preferred: (i) activity type; (ii) location; (iii) time of day (morning, afternoon or evening). The system provided the user with a list of activities matching the chosen criteria. The improved strategy resulted in an increase in task completion from 52% to 64%.

More recently, an evaluation of the TownInfo system compared a complete hand-coded strategy with a learned strategy (Lemon et al., 2006). The system allowed users to ask for information on hotels, bars and restaurants for a fictitious town depicted on a visual display. The learned strategy was previously used for a flight booking system but was mapped on to a tourist information task by changing system actions, prompts etc. The behaviour of the learned strategy remained the same. The learned and hand-coded strategies were compared by asking 18 participants to each perform 10 tourist information tasks (5 with the learned strategy, 5 with the hand-coded strategy). The evaluation results showed that users' perceived task completion of dialogues involving the learned strategy was comparable with the hand-coded one but required less time to complete, i.e. 3.3 system turns less per dialogue. The results also demonstrated that learned strategies could be ported between domains.

The present study differs from these systems in several important ways. Firstly, the ELVIS and NJFun systems use very small state spaces for learning – 18 and 62 unique states respectively. This makes learning much more feasible but also means that much fewer global strategies can be explored. This study makes use of state space that is much too large to represent in tabular form (i.e. $> 10^9$ state-action pairs), making traditional reinforcement learning techniques unworkable. However, this allows a global strategy to be learned. The TownInfo system also employs a very large state space, employing linear function approximation to generalise over the state space (Henderson et al., 2005)

Secondly, this study allows users to interact with a live flight booking system, with over 250,000 unique flight queries available. This has important consequences for confirmation strategies. The NJFun system acts as an interface to an experimental version of a real application, containing only "149 distinct database entries" (Singh et al., 2000, page 3). The ELVIS system is also experimental; it is not clear from the system description just how large the implemented email folders are. The TownInfo database contains details of only six hotels, six bars and six restaurants. The study acknowledged that the reporting of query results "would be more of a problem with a much larger database" (Lemon et al., 2006, section 6.3).

Thirdly, in terms of the strategies learned, all three studies use RL only to improve existing strategies. In the case of NJFun and ELVIS, learning was used to optimise some, but not all, of the choices available in the hand-coded strategies, such as the type of initiative to use (system versus mixed). In TownInfo, RL was used to improve a strategy that was first created by applying supervised learning methods to a corpus of human-computer dialogues. Therefore, this study is unique in terms of learning a complete dialogue strategy from scratch instead of improving an existing hand-coded or previously learned strategy.

Most importantly, the present study differs from all three previous studies in its absence of a training corpus. The strategies developed in each of these previous studies were based on corpora of human-computer dialogues. In the ELVIS and NJFun systems, corpora of 219 and 311 human-computer dialogues respectively were collected for the learning of an initial sub-strategy. The learned strategy for the TownInfo system was derived from the Communicator corpora, which took several months to collect, transcribe and annotate; it contains over 2000 human-computer dialogues.

This study is fundamentally different from the others since the omission of a training corpus dramatically reduces the development time. However, the question remains whether this approach will be successful with real users.

6.2 Experimental aims

This evaluation has two main aims. Firstly, to investigate whether evolutionary learning, in conjunction with a hand-coded simulated user, can generate a dialogue strategy that is positively evaluated by real users. More specifically, can the learned strategy achieve an acceptable level of performance using established objective and subjective evaluation metrics (e.g. Walker et al., 2000)?

The second aim is to examine whether the learned strategy compares favourably with one that is developed for the same task using the traditional test-and-refine method. How does the learned strategy compare with the hand-coded strategy using the same evaluation metrics? If the learned strategy performs as well as the hand-coded one but requires much less time to develop, this study will provide an important contribution to the field of dialogue strategy learning.

6.3 Experimental methodology

To evaluate the effectiveness of a learned strategy with real users, the following approach was taken. Firstly, a useful, realistic evaluation task was selected. Secondly, I implemented an end-to-end dialogue system, complete with speech and language-processing modules. The system's dialogue manager was designed to facilitate the easy integration of different dialogue strategies. To properly evaluate a learned strategy, I first designed a state-of-the-art hand-coded baseline strategy. This involved the selection of an appropriate set of dialogue acts and a state representation. The same dialogue acts and a subset of the state features were then used to generate a learned strategy. An evaluation framework was developed to allow the complete dialogue system to work interchangeably with the hand-coded and learned strategies. Using this framework, the performances of the hand-coded and learned strategies were compared. This comparison was based on an evaluation by human participants.

6.3.1 A flight booking system

The chosen evaluation system was based on the flight booking domain. This domain has been used to evaluate dialogue strategies in a number of large-scale systems and research programmes, most notably the Communicator project (Walker et al., 2000, 2001). This programme provided a common system architecture and evaluation framework, allowing a variety of commercial and research teams to compare the performance of both individual system modules and complete systems. The Communicator project served as a useful model for the present study.

This evaluation is based on the web-based flight booking system of FlyGlobeSpan, an Edinburgh-based airline. At the time of writing, FlyGlobeSpan serves over 50 unique routes, primarily between Scotland and Europe. At the time of evaluation, the winter schedule was in operation (October to March). For this evaluation, users were asked for four slot values: departure and destination cities (*depValue* and *destValue*) and dates for the outward and return journeys (*outValue* and *retValue*).

For this evaluation to be realistic, the SDS should exhibit many of the features that are typical of state-of-the-art spoken dialogue systems. In other words, it should allow users, with a variety of expertise, to accomplish the goal of booking a flight in a flexible manner, using a natural vocabulary. More concretely, the evaluation system should contain the following features:

- open speech recognition grammar
- highly intelligible and natural speech output
- natural language interpretation and generation
- connection to live flight database
- mixed-initiative, allowing users to be over-informative
- recognition threshold-based confirmation prompts (explicit and implicit)
- context-sensitive help facility
- allow users to end dialogue prematurely
- handoff to human operator when dialogue is not progressing
- interchangeable use of hand-coded and learned strategies

This flight booking system did not include a facility for constraint refinement. This is where a user can repeatedly refine the parameters of the database query in order to find the desired information. This is a complex task in itself (Demberg and Moore, 2006; Polifroni and Walker, 2006) and was beyond the scope of this evaluation. Therefore, in this evaluation system, a task was considered to be completed when the system retrieved the results from the database for the user's initial query.

6.3.2 System outline

The design of the implemented system followed the traditional architecture for spoken dialogue systems (section 2.2.2). In other words, the system comprised self-contained modules for: (i) speech recognition; (ii) speech synthesis; (iii) natural language parsing; (iv) natural language generation; (v) database connectivity; and (vi) dialogue management. All of the modules were implemented as Java objects using JDK 1.5.

This system integrated existing modules for speech recognition and synthesis with hand-coded modules implementing: language parsing, language generation, database connectivity and dialogue management. The speech recogniser used was the speaker-independent HMM-based Sphinx-4 system available from Carnegie-Mellon University (Lamere et al., 2003). The speech synthesiser was a unit selection-based synthesiser purchased from Cepstral Inc (Cepstral, 2007). The natural language parser consisted of a hand-coded set of pattern matching rules and some code to parse dates. Similarly, the language generator was a relatively simple template-based system, comprising a set of rules for combining canned text with slot values. The module for retrieving flight details was a small suite of webscraping programs that connected to the FlyGlobeSpan website.

Finally, the dialogue manager was made up of two sections: (i) a sub-module for updating the dialogue state according to the user's last utterance; (ii) two instances of a sub-module for choosing the next system dialogue act based on the current dialogue state (one was the hand-coded strategy, the other was a strategy learned by XCS). All of these modules were coordinated by the main evaluation program. For each dialogue, the evaluation program could control whether the dialogue manager should select the next system action using the hand-coded strategy or the learned one.

6.3.2.1 Dialogue acts

The chosen system and user dialogue acts are summarised in Table 6.1. They are based on the dialogue acts used in the previous chapter on large state-action spaces. The system dialogue acts allow the system to ask the user for the slot values or to confirm these values, either explicitly or implicitly. The system can restart or end the dialogue, give the user help or hand the user on to a human operator. Finally, the system can present the results of a user's database query.

System acts	User acts
GREETING	command(bye)
GOODBYE	command(request_help)
HANDOFF	command(request_repetition)
GIVE_HELP	command(restart)
RESTART	answer(yes)
REQUEST_INFO(depValue)	answer(no)
REQUEST_INFO(destValue)	provide_info(depValue) *
REQUEST_INFO(outValue)	provide_info(destValue) *
REQUEST_INFO(retValue)	provide_info(outValue) *
REQUEST_INFO_WITH_IMPCONF(depValue)	provide_info(retValue) *
REQUEST_INFO_WITH_IMPCONF(destValue)	unknown
REQUEST_INFO_WITH_IMPCONF(outValue)	
REQUEST_INFO_WITH_IMPCONF(retValue)	
EXPCONF(lo+med_confs)	* *multiple slot values can be*
DBASE_RESULTS	*provided in a single utterance*

Table 6.1: System and user dialogue acts for the evaluation system.

The user dialogue acts allow the user to terminate the dialogue, ask for help, ask the system to repeat its last utterance, start the dialogue from the beginning or provide slot information. The user can also confirm or reject a system's explicit confirmation of slot values. The user can provide the system with more than one slot value in a single utterance, including slot values that were not asked for by the system in order to reduce the length of the dialogue. The *unknown* act represents a user utterance that cannot be interpreted satisfactorily.

6.3.2.2 State representation

The state variables used to represent the dialogue state were based on literature relevant to developing flight booking systems and the state representation used in the previous chapter on large state-action spaces. In common with many information-seeking tasks, the slot values and associated recognition confidences were stored. For this application, the confidence value received from the speech recogniser was mapped on to one of four values: 'empty', 'lo', 'med', 'hi'. It is also usual to record the number of turns taken so far, the most recent user act and the previous system act. During the development of the hand-coded strategy, the penultimate system act was also used. A record of the number of times each slot was asked for or confirmed by the system was necessary to indicate that a dialogue was not progressing sufficiently, perhaps due to the persistent misrecognition of a particular word. In such a situation, the system should pass the user on to a human operator. The list of state variables and their possible values are given in Table 6.2.

State Variable	Possible values
turn_number	1..50
departure_value	'edinburgh, 'glasgow' etc.
destination_value	'barcelona', 'nice' etc.
outward_date_value	01/11/06 .. 31/03/07
return_date_value	01/11/06 .. 31/03/07
departure_confidence	empty, lo, med, hi
destination_confidence	empty, lo, med, hi
outward_date_confidence	empty, lo, med, hi
return_date_confidence	empty, lo, med, hi
departure_times_asked	1..10
destination_times_asked	1..10
outward_date_times_asked	1..10
return_date_times_asked	1..10
last_user_act	11 values (see Table 6.1)
last_system_act	15 values (see Table 6.1)
2nd_last_system_act	15 values (see Table 6.1)

Table 6.2: State representation for the evaluation system.

6.3.2.3 User vocabulary

The user vocabulary was based on an analysis of the transcriptions for the Communicator evaluations (Walker et al., 2001), in addition to a number of other words that seemed appropriate for the intended application. Naturally, the vocabulary included the names of the airports used by the FlyGlobeSpan web site. The user vocabulary also included the facility for specifying dates (e.g. "the tenth of March", "January fifteenth").

A series of tests was conducted to determine the appropriate size of the user vocabulary in order to minimize speech recognition errors. In the end, a corpus of 202 sentences was used to create a language model containing 389 trigrams, 398 bigrams and 179 unigrams. This was sufficient to create a keyword grammar, enabling the language parser to generate appropriate user dialogue acts. The recognition grammar was open, in the sense that all words in the recognition vocabulary could be recognised at any point in the dialogue. Examples of the user vocabulary and how they were parsed into user dialogue acts is given in Table 6.3:

User utterance	User dialogue act
"*I want to quit*"	command(bye)
"*I need some help*"	command(request_help)
"*Say that again*"	command(request_repetition)
"*I want to start over*"	command(restart)
"*yes, that's correct*"	answer(yes)
"*no, that's wrong*"	answer(no)
"*I'm flying from Edinburgh to Barcelona*"	provide_info(depValue='edinburgh', destValue='barcelona')

Table 6.3: Examples of user vocabulary and corresponding dialogue acts.

6.3.3 A hand-coded strategy

A hand-coded strategy was developed as a baseline for comparison with the learned strategy. The hand-coded strategy was tested and refined eight times until what was considered an optimal strategy had been developed. The strategy was based on the

requirements for system performance, outlined in section 6.3.1. The hand-coded strategy drives the system to elicit values from the user for the four slots. Where slot values do not have an associated high ('hi') recognition confidence, the system confirms these slot values – explicitly if the confidence is low ('lo') and implicitly if the confidence is medium ('med'). The system provides the user with context-sensitive help when asked by the user, giving examples of what the user might say. It allows the user to take the initiative by answering a different question to the one that was asked. The user can also give more information than was asked for, thereby reducing the length of the dialogue. Finally, the greeting prompt, "How can I help you today?", allows the user to control the flow of dialogue from the beginning . All of these features could be implemented using the chosen dialogue acts and state representation.

6.3.4 A learned strategy

For the purpose of learning a dialogue strategy, not all of the state variables listed in Table 6.2 are relevant. For example, the values for slots themselves are not required for learning; only the recognition confidences associated with them are required. Based on the results detailed in the previous chapter, the following variables were chosen for a learned strategy: (i) the four slot confidence variables; (ii) the four variables indicating how many times each slots was asked for or confirmed; (iii) the last user act; (iv) the last system act. This means that the size of the state space was $4^4 \times 10^4 \times 11 \times 15$ which is approximately 4.2×10^8. With 15 available system actions, this means that the size of the state-action space was 6.3×10^9. This is not an unusually large size for a state-action space for a simple slot-filling task. However, it is clearly much too large to enumerate all possible combinations. Consequently, traditional tabular reinforcement learning could not be applied. This issue is the motivation to employ a learning algorithm that can generalise over large state-action spaces (Henderson et al., 2005; Cuayáhuitl et al., 2006a).

This state representation was used to generate a dialogue strategy in conjunction with: (i) the set of system and user dialogue acts in Table 6.1; (ii) the reward function presented in Table 5.3; (iii) a stochastic, goal-directed simulated user similar to Figure 5.3. The learned strategy was described by 579 XCS condition-action rules. A module was written to input a state representation and to use the XCS rule set to generate a new system act. This module was integrated into the evaluation framework.

6.3.5 Experimental procedure

This experiment used a within-subjects design. Each participant was asked to perform three simple flight booking tasks for each strategy. The order of strategy presentation was counterbalanced to address the potential effect of order on the experimental results. Participants were randomly allocated to one of two order conditions (hand-coded then learned, learned then hand-coded). In the first two tasks, the goal was fixed; in the third task, the goal was left open to the participant (Figure 6.1). Each task was then repeated (one for each strategy), giving a total of six conversations.

---

1. You are spending your Christmas holidays in Florida this year. Please book a flight from Glasgow to Orlando, leaving on December 23rd and returning on January 10th.

2. You are attending a one-day conference at Barcelona in March. Please book a flight from Edinburgh to Barcelona, leaving on March 1st and returning on March 3rd.

3. You have won a free flight in FlyGlobeSpan's Monthly Draw. Please book any flight available in FlyGlobeSpan's winter schedule.

Figure 6.1: Evaluation task descriptions.

Before taking part in the evaluation, each participant was told that this was an evaluation of speech technology but was not told explicitly that dialogue strategies were being compared. This information was given to participants after the tasks had been completed. Participants were also asked to indicate their level of familiarity with speech or language processing technology and also to characterise their accent. Non-native English speakers were not included in the sample. Lastly, participants were assured that their data would be anonymised.

For each dialogue, the user utterances were recorded. In addition, a log file was generated for each dialogue. For each system or user turn, the folowing information was recorded: the turn number, the system or user utterance, relevant dialogue state variables and the cumulative elapsed time in the dialogue. An example of a logged dialogue turn is given in Figure 6.2.

```
            Turn:  7
            Type:  System
            Text:  Okay, please give me the date you wish to leave.
     System Act:  REQUEST_INFO(outValue)
  Last System Act  REQUEST_INFO(retValue)
     Slot Values:  'edinburgh', 'barcelona', 'empty', 'empty'
      Slot Confs:  'hi', 'hi', 'none', 'none'
Slot Times Asked:  1, 1, 1, 0
   Last User Act:  provide_info(destValue='barcelona')
     Time (secs):  25.19
```

Figure 6.2: Example of a logged dialogue turn.

After completing each task, participants were asked to complete a short questionnaire. They were first asked whether they were able to complete the task successfully. This response measures Perceived Task Completion (PTC). In contrast, Actual Task Completion (ATC) is a measure of whether the system answered the query indicated in the task. Participants were then asked to indicate their level of agreement with five statements using a Likert scale, ranging from 1 (strongly disagree) to 5 (strongly agree). These statements were based on subjective measures of user satisfaction used in the Communicator evaluations (Walker et al., 2000). The measures were: Task Ease, TTS Performance, User Expertise, Expected Behaviour and Future Use (Table 6.4).

Statement	Metric
It was easy to get the information I wanted.	Task Ease
I found the system easy to understand.	TTS Performance
I knew what I could say or do at each point in the dialogue.	User Expertise
The system worked the way I expected it to.	Expected Behaviour
Based on this experience, I would use this system regularly.	Future Use

Table 6.4: User satisfaction metrics.

6.4 Experimental results

The participant sample was made up of eight males and eight females. In terms of accent, eight were British, six were North American and two were Australasian. Of the 16 participants, six had no experience in the use of speech or language processing technology. Informal feedback from participants suggested that the performance of the speech recogniser was much better with North American participants. This is not surprising since the speech recogniser's acoustic models were trained on American voices. It should be noted that there was little difference between participants in the two order conditions in terms of accent, gender and previous experience. Each of the 16 participants attempted to complete their six tasks, giving a total 96 logged dialogues. A transcription of one dialogue (with dialogue acts) is given in Figure 6.3.

Although participants were informed that a help facility was available, no participants asked for help. Also, no users asked to restart the dialogue. Nine of the dialogues were stopped by participants before the task was completed. In four cases, this was by accident, due to a false recognition of a termination keyword ("finish", "stop", "quit", "exit"). Five dialogues were terminated by users who were frustrated by poor speech recognition. One of the dialogues was handed off by the dialogue system.

Preliminary statistical analysis examined whether there were gender differences which would need to be controlled for in the main analysis. A series of independent t-tests revealed no differences between male and female users. Thus, gender was not included in any subsequent analysis. A series of paired t-tests were then conducted to examine differences between the two systems for each of the three tasks.

6.4.1 Task completion

Table 6.5 shows the descriptive statistics for the two systems in terms of how users evaluated each of the three tasks on all of the measures. There were no significant differences between the hand-coded and learned strategy in either Actual or Perceived Task Completion. Both systems performed well on both Actual and Perceived Task Completion. Across all three tasks, there was an average of 77% Actual Task Completion (ATC) for the hand-coded strategy compared with 81% for the learned strategy, but this difference was not statistically significant ($t(15) = -1.00$, $p > .30$).

System: **Welcome to FlyGlobeSpan's booking system. How can I help you today?**
[GREETING]

User: **I want to book a flight.**
[unknown]

System: **Please give the name of the city that you are departing from.**
[REQUEST_INFO(depVal)]

User: **Edinburgh.**
[provide_info(depVal='edinburgh')]

System: **So you want to travel from edinburgh. What city are you travelling to?**
[REQUEST_INFO_WITH_IMPCONF(destVal)]

User: **Geneva.**
[provide_info(destVal='geneva')]

System: **Okay, please give me the date you wish to leave.**
[REQUEST_INFO(outVal)]

User: **December 24th.**
[provide_info(outVal='24/12/2006')]

System: **And finally, on what date do you wish to return?**
[REQUEST_INFO(retVal)]

User: **January 6th.**
[unknown]

System: **Could you please repeat that?**
[REQUEST_INFO(retVal)]

User: **January 6th.**
[provide_info(retVal='06/01/2007')]

System: Okay, accessing the database. <DATBASE ACCESS>.
Your outward flight on Sunday, December 24, leaves at 08:30.
Your return flight on Saturday, January 06, leaves at 11:40.
The total cost of your flight is 219 pounds. Goodbye.
[DBASE_RESULTS]

Figure 6.3: Example dialogue from the system evaluation.

Users' Perceived Task Completion (PTC) for the hand-coded strategy was found to be 92% across all three tasks while for the learned strategy it was found to be 88%, $t(15) = 1.00$, $p > .30$. Further analysis for each of the individual tasks also revealed no significant differences.

A similar pattern was revealed for the amount of time and number of turns taken for each task. There was no significant difference between the average time taken by the hand-coded strategy across all three tasks (M = 61.6 secs, SD = 9.3) and that taken by the learned strategy (M = 60.7 secs, SD = 6.4), $t(15) = 0.4$, $p > .70$). There was also no difference in the average number of turns taken between the hand-coded strategy (M = 13.1, SD = 3.2) and the learned strategy (M = 13.0, SD = 2.0), $t(15) = 0.2$, $p > .80$). Separate analyses by task revealed no further differences. It is very encouraging that the learned strategy performed as well as the hand-coded one with respect to these objective measures.

	System A (hand-coded)			**System B (learned)**		
	Task 1	Task 2	Task 3	Task 1	Task 2	Task 3
Time (secs.)	58.2 (16.4)	62.1 (11.9)	64.5 (11.2)	59.8 (10.1)	56.0 (16.0)	66.3 (20.8)
Turns	11.8 (4.6)	13.8 (5.3)	13.9 (4.1)	12.5 (2.4)	12.0 (5.1)	14.6 (5.5)
ATC (%)	68.8 (47.9)	81.3 (40.3)	81.3 (40.3)	75.0 (44.7)	81.3 (40.3)	87.5 (34.2)
PTC (%)	93.8 (25.0)	93.8 (25.0)	87.5 (34.2)	93.8 (25.0)	81.3 (40.3)	87.5 (34.2)
Task Ease	3.8 (1.2)	3.8 (1.1)	3.7 (1.1)	4.1 (1.0)	4.1 (1.2)	3.7 (0.9)
TTS	4.4 (1.1)	4.8 (0.5)	4.7 (0.5)	4.6 (1.0)	4.8 (0.4)	4.6 (0.6)
Expertise	4.0 (1.0)	4.3 (0.5)	4.3 (0.5)	4.4 (0.7)	4.4 (0.6)	4.3 (0.7)
Expected	3.5 (1.0)	3.8 (1.0)	3.4 (0.9)	3.8 (1.0)	3.9 (1.0)	3.7 (0.9)
Future Use	3.5 (1.2)	3.7 (1.0)	3.6 (1.1)	4.1 (0.6)	3.9 (1.1)	3.7 (1.0)
User Sat.	3.9 (0.9)	4.1 (0.5)	3.9 (0.5)	4.2 (0.5)	4.3 (0.7)	4.0 (0.5)

Table 6.5: Summary of evaluation results.

6.4.2 User satisfaction

Analyses examined all five aspects of user satisfaction separately. First, there was no significant difference in the average perceived ease of use between the hand-coded strategy (M = 3.8, SD = 0.8) and the learned strategy (M = 4.0, SD = 0.6), $t(15) = -1.58$, $p > .25$).

Second, the hand-coded strategy (M = 4.6, SD = 0.5) and the learned strategy (M = 4.7, SD = 0.5) were both rated positively and did not differ significantly on TTS, t(15) = -0.12, p>.85.

Third, there was no difference between the two systems on expected behaviour, t(15) = -1.45, p>.15, with the hand-coded strategy rated 3.6 on average (SD = 0.5) and the learned strategy rated 3.8 (SD = 0.6).

On user expertise, the two systems were rated positively (hand-coded M = 4.2, SD = 0.5, learned M = 4.4, SD = 0.6) and again no significant difference was indicated, t(15) = -1.52, p>.10.

Finally, there was a significant difference between the two systems on the extent to which users would use the system in the future. The learned strategy was rated more positively (M = 3.9, SD = 0.8) compared with the hand-coded strategy (M = 3.6, SD = 0.8) t(15) = -3.65, p<.01).

Further analysis by task revealed a trend approaching significance for the learned strategy to perform better on future use for Tasks 1 (p<.06) and 2 (p<.17). No further differences between the two systems were indicated.

6.5 Discussion

I cannot make statistical comparisons with the results of this evaluation and the evaluation of systems reported in section 6.1. However, inspection of mean scores for the subjective evaluation metrics suggest that the performance of this system compares favourably to those of other systems. For example, the overall user satisfaction score for the learned strategy in the ELVIS evaluation (Walker, 2000) was 31.7 of a possible 40 (= 79%); for the NJFun system (Litman et al., 2000), the satisfaction score was 13.29/20 (= 66 %); for the TownInfo evaluation (Lemon et al., 2006), the mean user satisfaction score was 2.67/5.0 (= 53 %). The present study was the highest, with a mean user satisfaction score of 4.16/5.0 (= 83 %).

Looking at task completion, the Actual Task Completion (ATC) scores for the hand-coded and learned strategies in NJFun were 52% and 64% respectively; the mean ATC scores for the hand-coded and learned strategies in this study were 77% and 81% .

In the TownInfo evaluation the Perceived Task Completion (PTC) scores for the hand-coded and learned strategies were 67.6% and 81.8% respectively. The PTC scores in this study were 92% and 88% respectively. No other task completion results have been reported for these systems.

Comparing this evaluation with an evaluation of the Communicator systems (Walker et al., 2001), the results remain positive. The June 2000 Data Collection of the Communicator programme comprised 662 dialogues, spread across nine systems. In this evaluation the user satisfaction score was computed as the total of the five metrics listed in Table 6.4. The reported median user satisfaction for the nine participating systems ranged from 5.0 to 15.5 (maximum = 25). The median user satisfaction for the hand-coded and learned strategies in the present study were 18.9 and 21.0 respectively. Clearly, the Communicator systems are more sophisticated in terms of the range of dialogue acts that are available and the tasks that can be completed. Additionally, it should be stressed that these comparisons are not based on systematic statistical analysis. Nevertheless, the results of this evaluation are very encouraging.

More importantly, these evaluation results substantiate the claim that evolutionary reinforcement learning, in conjunction with a hand-coded simulated user, can generate a dialogue strategy that performs as well as a hand-coded one. In terms of objective measures – number of dialogue turns, dialogue duration and Actual Task Completion – there is no significant difference between the learned and hand-coded strategies. In terms of user satisfaction, one significant difference was found: the learned strategy was rated more positively in terms of anticipated future use ($p < .01$). This appears to be good news for the learned strategy. Perhaps there are some aspects of user satisfaction that are not being properly assessed by the chosen evaluation metrics.

The fact that the learned strategy performed as well as the hand-coded one in this evaluation is an important result. Most of the recent research on the learning of dialogue strategies have based user simulations on training corpora. Clearly, if such data is available then it makes sense to utilise it. However, the issue becomes what should be done if this data does not exist. Hopefully, a lot of novel SDSs will be developed in the future, with no pre-existing corpora available. If a corpus needs to be collected in order to create a simulated user, the intended reduction in development time – through the use of learning algorithms – will probably not occur. This evaluation has demonstrated that the collection of a training corpus may not be necessary.

6.6 Summary

In this chapter, I have presented an evaluation by real users of a dialogue strategy generated by evolutionary reinforcement learning in conjunction with a hand-coded simulated user. Both the learned strategy and a state-of-the-art hand-coded strategy were integrated into an end-to-end spoken dialogue system. The dialogue system allowed real users to make flight enquiries using a live database for an Edinburgh-based airline. The performance of the learned and hand-coded strategies were compared. The evaluation results showed that the learned strategy performed at a high level and as well as the hand-coded one (e.g. 81% and 77% task completion respectively). This is an important result because the learned strategy took only two days to develop while the hand-coded one was refined over a period of two weeks. Therefore, the use of XCS, an evolutionary reinforcement learning algorithm, together with a hand-coded simulated user, is a viable approach to developing spoken dialogue strategies.

Chapter 7

Conclusion

7.1 Summary of contributions

Throughout this thesis, I have argued that the learning of spoken dialogue strategies is motivated by the desire to "build good dialogue strategies quickly". The research presented here has combined an evolutionary learning algorithm with hand-coded simulated users to accomplish this goal. In the process, this thesis has made several contributions to the research field. They are summarised below. It should also be noted that this research was undertaken in the context of spoken dialogue systems; the research could equally be applied to text-based dialogue systems.

Evolutionary learning. This is the first application of evolutionary algorithms to the problem of learning spoken dialogue strategies (Toney et al., 2006a,b). All previous work in learning strategies has been based on reinforcement learning techniques. Why use a different algorithm? There remains a number of important challenges associated with strategy learning (see section 2.4.7). It is suggested that one particular evolutionary algorithm, XCS, possesses a number of useful properties (e.g. rule-based representation, ability to generalise over large state spaces) that make it an appealing candidate for learning dialogue strategies. Furthermore, during the course of this thesis, contact has been made with a number of research groups specialising in evolutionary algorithms. The work undertaken has generated a great deal of interest among some of these groups and has resulted in a proposal for collaborative research.

Large state-action spaces. A key feature of the XCS algorithm is that it facilitates the learning of strategies that require large state-action representations. In the context of reinforcement learning problems, "large" means a state-action space that would require greater than, say, one million (10^6) elements when represented in tabular form. However, real-life SDSs usually employ state-action representations that are much larger than this. This is arguably the most important challenge to developing dialogue strategies for realistic tasks. Therefore, for strategy learning, some method of generalising over a large state-action space is required. Thus far, only one study has reported a technique for doing this (Henderson et al., 2005). However, this work required a training corpus since it employed a hybrid supervised/reinforcement learning framework. Therefore, this thesis is unique in learning realistic strategies for large state-action spaces without the use of a corpus.

Hand-coded simulated users. A crucial feature of this thesis was the use of hand-coded simulated users. Although the simulated behaviour was stochastic and goal-directed, it was not created through the use of a training corpus. I have argued that achieving good results with hand-coded users is essential for the rapid development of dialogue strategies. I have also argued that hand-coded users are sufficient for learning useful dialogue strategies. I have substantiated this claim by using a hand-coded simulated user to generate a dialogue strategy which was positively evaluated by real users.

Evaluation by real users. Ultimately, the effectiveness of any technique for generating dialogue strategies can only be properly assessed through an evaluation by real users. This thesis has presented such an evaluation. This required the development of a complete end-to-end dialogue system, which was a time-consuming task. Nevertheless, it was an essential part of this thesis; the empirical results demonstrated that evolutionary learning, in conjunction with a hand-coded simulated user, performed as well as a state-of-the-art hand-coded one. Furthermore, both the hand-coded and learned strategies in this evaluation compared favourably with other systems that used the same evaluation metrics.

This thesis also reported on the three previous studies that evaluated learned strategies with real users. This study differed from those in several ways. Firstly, only one of

the previous studies employed large state-action spaces. Secondly, this study was the only one to make use of a large live database. Thirdly, the previous studies evaluated strategies that were either optimised versions of existing hand-coded strategies or an improved version of a strategy created by supervised learning. Therefore, this is the first evaluation by real users of a strategy that was learned from scratch. Lastly, the evaluation results of this study compared very favourably with the previous studies, both in terms of task completion and user satisfaction.

7.2 Future research directions

The research outlined in this thesis provides a solid basis for continued research into the rapid development of learned dialogue strategies. However, it is not yet clear whether learning techniques are a viable alternative to the traditional test-and-refine model of strategy development (particularly in a commercial environment). If we are to create a software tool for automatically generating dialogue strategies, a number of other technical challenges must be addressed. A summary of what I perceive to be the main challenges is given below. Finally, it is important to emphasise that ongoing research within the XCS community is tackling many of these issues (Bull, 2004; Bull and Lanzi, 2007). I believe that this makes XCS an excellent platform for ongoing research on learning dialogue strategies. It is hoped that a XCS-based tool can be developed for implementing sophisticated spoken dialogue systems.

Complex tasks. So far, much of the research on strategy learning has focused on information-seeking (slot-filling) tasks, although some work has looked at tutorial systems (Litman and Forbes-Riley, 2006). It remains to be seen whether strategy learning can be usefully applied to more complex tasks, requiring more sophisticated collaboration, negotiation and planning between system and user. Moreover, very little work has been undertaken with respect to hierarchical task structures (but see Cuayáhuitl et al., 2006b). In time, it is expected that continued research will tackle increasingly more complex tasks. With respect to information-seeking systems, I would suggest that learning a dialogue strategy for a full Communicator system – with a very large state-action space and a range of nested tasks – could be considered a reasonable indicator that strategy learning techniques have a future.

It would also be instructive to investigate the effect of learning dialogue strategies using XCS with increasingly larger state-action spaces. In other words, it is important to establish whether there is an upper bound on learning useful dialogue strategies. To examine this, additional state variables could be incorporated into the state representation. Equally, the range of possible values for existing state variables, could be increased. For instance, the present speech recognition confidence values of 'empty', 'lo', 'med', 'hi' could be replaced by the integer range 0..100. Furthermore, it would be useful to explore the relationship between the size of the state space and the time taken to learn a dialogue strategy. Thus far, analysis of the time complexity of XCS has been concerned only with classification (single-step) rather than sequential (multi-step) tasks (Wilson, 1998; Butz et al., 2004). In terms of state space size, a binary classification task with 2^{37} unique possible states has been successfully addressed using XCS (Butz et al., 2001).

Strategy summarisation. It is important for human developers to understand, and learn from, the strategies generated by machine learning methods. However, summarising learned strategies for interpretation by developers remains difficult. This is an issue for the RL-based methods for strategy learning and for the present XCS-based approach. In the evaluation system, the final rule set consisted of 579 condition-action rules. While this may represent a complete mapping of the state-action space, it is still much too large for inspection by a developer. Some method of enumerating the most useful rules of the learned strategy is required. Recent research in the XCS community has successfully produced compact rule sets for classification problems (Fu and Davis, 2002). As yet, the same results have not been achieved with sequential tasks.

Reward function selection. For an information-seeking task, it is reasonably straightforward, in principle, to select an appropriate reward model for learning dialogue strategies: reward slot-filling, discourage excessive turn-taking etc. However, the selection of reward functions for more complex tasks has received very little attention so far. This issue may prove to be a difficult stumbling block in the pursuit of learning dialogue strategies. Therefore, it is important to pursue an increased understanding of the effects of reward functions. Furthermore, it may be possible to implement a method for automatically testing and selecting useful reward functions.

Partial observability. The issue of partial observability, arising from speech recognition errors, was described in section 2.4.7. This refers to the fact that, for a given utterance, a dialogue manager cannot be certain of what was said by a user. Instead, dialogue state representations typically record a recognition confidence score for each slot value. However, several studies have shown that maintaining a distribution of possible slot values, and associated recognition confidence scores, can improve the performance of strategy learners. Furthermore, the advantage gained by modelling the partial observability increases with a system's recognition error rate. However, the computational cost associated with modelling this partial observability while learning dialogue strategies is very high. Consequently, only small state representations have been used (e.g. Williams and Young, 2006). Further research is needed to fully exploit uncertainty models with strategy learning.

Convergence detection. In contrast to many sequential decision problems, it is difficult to precisely define an optimal dialogue strategy. Studies on dialogue strategy learning typically present the performance of a strategy learner as a graph of some reward measure over a fixed number of learning episodes (e.g. 100,000). Convergence towards an optimal solution is indicated visually by the flattening of the learning curve. If we are to build a software tool for learning dialogue strategies, it is important that the strategy learner terminates automatically when an optimal solution has been generated. This is a goal that is very easy to describe qualitatively but remains difficult to implement in practice.

Comparison with other methods. It would be informative to quantitatively compare the time taken to generate a strategy using the traditional test-and-refine (hand-coding) approach with the use of evolutionary learning. It would also be useful to conduct a comparative study of the learning techniques reported so far for learning dialogue strategies (e.g. tabular RL, RL with function approximation, XCS). Such a study should report on the space and time complexities of each technique and and on the efficacy of the generated strategies. Ideally, this would entail an evaluation of the learned strategies by real users. Finally, a quantitative study of the cost and benefit of hand-coded versus data-driven simulated users should also be conducted.

Bibliography

Ai, H. and Litman, D. (2006). Comparing real-real, simulated-simulated, and simulated-real spoken dialogue corpora. In *AAAI Workshop on Statistical and Empirical Approaches for Spoken Dialogue Systems*, Boston, USA.

Allen, J., Byron, D., Dzikovska, M., Ferguson, G., Galescu, L., and Stent, A. (2000). An architecture for a generic dialogue shell. *Natural Language Engineering*, 6(3–4):213–228.

Allen, J. and Ferguson, G. (2002). Human-Machine Collaborative Planning. In *3rd International NASA Workshop on Planning and Scheduling for Space*, Houston, USA.

Allen, J., Miller, B., Ringger, E., and Sikorski, T. (1996). A Robust System for Natural Spoken Dialogue. In *34th Annual Meeting of the Association of Compuational Linguistics (ACL)*, Santa Cruz, USA.

Andreou, A., Georgopoulos, E., and Likothanassis, S. (2002). Exchange-rates forecasting: A hybrid algorithm based on genetically optimized adaptive neural networks. *Computational Economics*, 20(3):191–210.

Aust, H. and Oerder, M. (1995). Dialogue control in automatic inquiry systems. In *ESCA Workshop on Spoken Dialogue Systems*, Vigsø, Denmark.

Aust, H., Oerder, M., Seide, F., and Steinbiss, V. (1995). The Philips automatic train timetable information system. *Speech Communication*, 17:249–262.

Baxter, J. and Bartlett, P. (2001). Infinite-Horizon Policy-Gradient Estimation. *Journal of Artificial Intelligence Research*, 15:319–350.

Bennett, C., Llitjós, S. A. F., Rudnicky, A., and Black, A. (2002). Building VoiceXML-Based Applications. In *7th International Conference on Spoken Language Processing (ICSLP)*, Denver, USA.

Berry, M. J. A. and Linoff, G. S. (2004). *Data Mining Techniques: For Marketing, Sales, and Customer Relationship Management (2nd edition)*. Hungry Minds Inc.

Bloom, B. (1984). The 2 sigma problem: The search for a method of group instruction as effective as one-to-one tutoring. *Educational Researcher*, 13:4–16.

Bos, J., Klein, E., Lemon, O., and Oka, T. (2003). DIPPER: Description and Formalisation of an Information-State Update Dialogue System Architecture. In *4th SIGdial Workshop on Discourse and Dialogue*, Sapporo, Japan.

Brøndsted, T., Larsen, L., Manthey, M., McKevitt, P., Moeslund, T., and Olesen, K. (2002). Developing intelligent multimedia applications. In *Multimodality in Language and Speech Systems*, pages 149–171. Kluwer Academic Publishers, Dordrecht, The Netherlands.

Bull, L., editor (2004). *Applications of Learning Classifier Systems*. Springer.

Bull, L. and Lanzi, P. L. (2007). Special issue on learning classifier systems. *Journal of Natural Computing*. To appear.

Butz, M., Goldberg, D., and Lanzi, P. (2004). Bounding Learning Time in XCS. Technical Report No. 2004003 IlliGAL, University of Illinois at Urbana-Champaign.

Butz, M., Kovacs, T., Lanzi, P. L., and Wilson, S. (2001). How XCS Evolves Accurate Classifiers. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 927–934, San Francisco, USA.

Butz, M. and Wilson, S. (2002). An algorithmic description of XCS. *Soft Computing*, 6:144–153.

Cavazza, M. (2001). An empirical study of speech recognition errors in a task-oriented dialogue system. In *2nd SIGdial Workshop on Discourse and Dialogue*, Aalborg, Denmark.

Cepstral (2007). http://www.cepstral.com.

Chung, G. (2004). Developing a Flexible Spoken Dialog System Using Simulation. In *Proceedings of the Association of Computational Linguistics (ACL)*, Barcelona, Spain.

Clark, H. (1996). *Using Language*. Cambridge University Press, Cambridge, UK.

Crook, P. and Hayes, G. (2003a). Active perception in navigation of partially observable grid worlds. In *6th European Workshop on Reinforcement Learning (EWRL)*, Nancy, France.

Crook, P. and Hayes, G. (2003b). Learning in a State of Confusion: Perceptual Aliasing in Grid World Navigation. In *Towards Intelligent Mobile Robots (TIMR)*, Bristol, UK.

Cuayáhuitl, H., Renals, S., Lemon, O., and Shimodaira, H. (2005). Human-computer dialogue simulation using hidden markov models. In *9th IEEE Automatic Speech Recognition and Understanding Workshop*, San Juan, Puerto Rico.

Cuayáhuitl, H., Renals, S., Lemon, O., and Shimodaira, H. (2006a). Learning multi-goal dialogue strategies using reinforcement learning with reduced state-action spaces. In *9th International Conference on Spoken Language Processing (ICSLP)*, Pittsburgh, USA.

Cuayáhuitl, H., Renals, S., Lemon, O., and Shimodaira, H. (2006b). Reinforcement learning of dialogue strategies with hierarchical abstract machines. In *1st IEEE/ACL 2006 Workshop on Spoken Language Technology*, Palm Beach, Aruba.

De Jong, K. and Spears, W. (1992). A Formal Analysis of the Role of Multi-Point Crossover in Genetic Algorithms. *Annals of Mathematics and Artificial Intelligence*, 5(1):1–26.

Demberg, V. and Moore, J. (2006). Information Presentation in Spoken Dialogue Systems. In *11th Conference of the European Chapter of the Association for Computational Linguistics (EACL)*, Trento, Italy.

Denecke, M., Dohsaka, K., and Nakano, M. (2004). Fast Reinforcement Learning of Dialogue Policies Using Stable Function Approximation. In *1st International Joint Conference on Natural Language Processing*, pages 1–11, Hainan Island, China.

Eckert, W., Levin, E., and Pierracini, R. (1998). Automatic evaluation of spoken dialogue systems. In *2nd Workshop on the Formal Semantics and Pragmatics of Dialogue*, Twente, The Netherlands.

Ehrlich, U. (1999). Task Hierarchies Representing Sub-Dialogs in Speech Dialog Systems. In *6th European Conference on Speech Communication and Technology*, Budapest, Hungary.

English, M. and Heeman, P. (2005). Learning mixed initiative dialog strategies by using reinforcement learning on both conversants. In *Conference on Empirical Methods in Natural Language Processing*, Vancouver, Canada.

Fogel, L., Owens, A., and Walsh, M. (1966). *Artificial Intelligence through Simulated Evolution*. Wiley Publishing, New York.

Frampton, M. and Lemon, O. (2005). Reinforcement learning of dialogue strategies using the user's last dialogue act. In *IJCAI Workshop on Knowledge and Reasoning in Practical Dialogue Systems*, Edinburgh, UK.

Frampton, M. and Lemon, O. (2006). Learning more effective dialogue strategies using limited dialogue move features. In *Proceedings of the ACL*, Sydney, Australia.

Fraser, N. and Gilbert, N. (1991). Simulating speech systems. *Computer Speech and Language*, 5:81–99.

Fu, C. and Davis, L. (2002). A modified classifier system compaction algorithm. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 920–925, New York, USA.

Georgila, K., Henderson, J., and Lemon, O. (2005). Learning User Simulations for Information State Update Dialogue Systems. In *9th European Conference on Speech Communication and Technology (INTERSPEECH)*, Lisbon, Portugal.

Goddeau, D. and Pineau, J. (2000). Fast Reinforcement Learning of Dialog Strategies. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Istanbul, Turkey.

Goldberg, D. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, Reading, MA, USA.

Goldberg, D. and Deb, K. (1991). A comparitive analysis of selection schemes used in genetic algorithms. In Rawlins, G., editor, *Foundations of Genetic Algorithms*, pages 69–93. Morgan Kaufmann.

Gorrell, G., Lewin, I., and Rayner, M. (2002). Adding Intelligent Help to Mixed Initiative Spoken Dialogue Systems. In *International Conference on Spoken Language Processing (ICSLP)*, Denver, USA.

Grefenstette, J., Ramsey, C., and Schultz, A. (1993). Learning sequential decision rules using simulation models and competition. *Machine Learning*, 5:355–381.

Grosz, B. and Sidner, C. (1986). Attentions, intentions and the structure of discourse. *Computational Linguistics*, 12:175–204.

Haykin, S. (1999). *Neural Networks: A Comprehensive Foundation (2nd edition)*. Pearson Education.

Heeman, P., Johnston, M., Denney, J., and Kaiser, E. (1998). Beyond Structured Dialogues: Factoring out Grounding. In *International Conference on Spoken Language Processing (ICSLP)*, Sydney, Australia.

Henderson, J., Lemon, O., and Georgila, K. (2005). Hybrid reinforcement/supervised learning for dialogue policies from COMMUNICATOR data. In *IJCAI Workshop on Knowledge and Reasoning in Practical Dialogue Systems*, Edinburgh, UK.

Holland, J. (1975). *Adaptation in natural and artificial systems*. University of Michigan Press, Ann Arbor, MI, USA.

Holland, J. (1976). Adaptation. In Rosen, R. and Snell, F., editors, *Progress in theoretical biology*. Plenum, New York.

Holland, J. and Reitman, J. (1978). Cognitive systems based on adaptive algorithms. In Waterman, D. A. and Hayes-Roth, F., editors, *Pattern-Directed Inference Systems*, pages 313–329. Academic Press.

Hone, K. and Baber, C. (2001). Designing habitable dialogues for speech-based interaction with computers. *International Journal of Human-Computer Studies*, 54(4):637–662.

Ishibuchi, H. and Nakashima, T. (2000). Linguistic rule extraction by genetics-based machine learning. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 195–202, Las Vegas, USA.

Jurafsky, D. and Martin, J. (2000). *Speech and Language Processing. An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition*. Prentice Hall, New Jersey, USA.

Kaelbling, L., Littman, M., and Cassandra, A. (1998). Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101:99–134.

Kaelbling, L., Littman, M., and Moore, A. (1996). Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research*, 4:237–285.

Klemmer, S., Sinha, A., Chen, J., Landay, J., Aboobaker, N., and Wang, A. (2000). Suede: a Wizard of Oz prototyping tool for speech user interfaces. In *13th Annual ACM Symposium on User Interface Software and Technology*, New York, USA.

Kovacs, T. (1997). XCS Classifier System Reliably Evolves Accurate, Complete, and Minimal Representations for Boolean Functions. In Roy, Chawdhry, and Pant, editors, *Soft Computing in Engineering Design and Manufacturing*, pages 59–68. Springer-Verlag.

Kovacs, T. (2002). *A Comparison of Strength and Accuracy-Based Fitness in Learning Classifier Systems*. PhD thesis, Birmingham University.

Koza, J. (1992). *Genetic Programming: On the programming of computers by means of natural selection*. MIT Press, Cambridge, MA, USA.

Koza, J., Keane, M., Streeter, M., Mydlowec, W., Yu, J., and Lanza, G. (2003). *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers.

Lamere, P., Kwok, P., Walker, W., Gouvea, E., Singh, R., Raj, B., and Wolf, P. (2003). Design of the CMU Sphinx-4 decoder. In *8th European Conference on Speech Communication and Technology (EUROSPEECH)*, Geneva, Switzerland.

Lanzi, P. L. (2002). Learning Classifier Systems from a Reinforcement Learning Perspective. *Soft Computing*, 6(3):162–170.

Lanzi, P. L. (2006). Personal communication. July, 2006.

Lanzi, P. L., Loiacono, D., Wilson, S., and Goldberg, D. (2005). XCS with computed prediction in multistep environments. In *Genetic And Evolutionary Computation Conference (GECCO)*, Washington, D.C., USA.

Lanzi, P. L., Stolzmann, W., and Wilson, S., editors (2000). *Learning Classifier Systems. From Foundations to Applications*. Springer-Verlag, Berlin.

Lanzi, P. L. and Wilson, S. (2000). Toward optimal classifier system performance in non-Markov environments. *Evolutionary Computation*, 8(4):393–418.

Larsson, S. and Traum, D. (2000). Information state and dialogue management in the TRINDI dialogue move engine toolkit. *Natural Language Engineering*, 6(3–4):323–340.

Lecœuche, R. (2001). Learning optimal dialogue management rules by using reinforcement learning and inductive logic programming. In *2nd Meeting of the North American Chapter of the Association of Computational Linguistics*, Pittsburgh, USA.

Lemon, O., Bracy, A., Gruenstein, A., and Peters, S. (2001). The WITAS Multi-modal Dialogue System I. In *7th European Conference on Speech and Communication Technology (EUROSPEECH)*, Aalborg, Denmark.

Lemon, O., Georgila, K., and Henderson, J. (2006). Evaluating Effectiveness and Portability of Reinforcement Learned Dialogue Strategies with real users: the TALK TownInfo Evaluation. In *1st IEEE/ACL 2006 Workshop on Spoken Language Technology*, Palm Beach, Aruba.

Levin, E., Pieraccini, R., and Eckert, W. (2000). A stochastic model of human-machine interaction for learning dialogue strategies. *IEEE Transactions on Speech and Audio Processing*, 8(1):11–23.

Lin, B. and Lee, L. (2001). Computer-Aided Analysis and Design for Spoken Dialogue Systems Based on Quantitative Simulations. *IEEE Transactions on Speech and Audio Processing*, 9(5):534–548.

Litman, D. (2002). Adding Spoken Dialogue to a Text-based Tutorial Dialogue System. In *ITS Workshop on Empirical Methods for Tutorial Dialogue Systems*, Biarritz, France.

Litman, D. and Forbes-Riley, K. (2006). Correlations betweeen dialogue acts and learning in spoken tutoring dialogues. *Natural Language Engineering*, 12(2):161–176.

Litman, D. and Hirschberg, J. (2006). Characterizing and Predicting Corrections in Spoken Dialogue Systems. *Computational Linguistics*, 32(3):417–438.

Litman, D., Kearns, M., Singh, S., and Walker, M. (2000). Automatic optimization of dialogue management. In *18th International Conference on Computational Linguistics (COLING)*, Saarbrucken, Germany.

López-Cózar, R., De la Torre, A., Segura, J., Rubio, A., and Sánchez, V. (2002). Testing dialogue systems by means of automatic generation of conversations. *Interacting with Computers*, 14(5):521–546.

Lucas, B. (2000). VoiceXML for Web-based Distributed Conversational Applications. *Communications of the ACM*, 43(9):53–57.

Mahadevan, S. (1996). Optimality Criteria in Reinforcement Learning. In *AAAI Fall Symposium on Learning Complex Behaviors for Intelligent Adaptive Systems*, Boston, USA.

McInnes, F., Nairn, I., Attwater, D., and Jack, M. (1999). Effects of prompt style on user responses to an automated banking service using word-spotting. *BT Technology Journal*, 17(1):160–171.

McRoy, S. (1998). Achieving Robust Human-Computer Communication. *International Journal of Human-Computer Studies*, 48(5):681–704.

McTear, M. (2004). *Spoken Dialogue Technology: Towards the Conversational User Interface*. Springer.

Mitchell, M. (1996). *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA.

Montana, D. and Davis, L. (1989). Training feedforward networks using genetic algorithms. In *12th International Joint Conference on Artificial Intelligence (IJCAI)*, Detroit, USA.

Moriarty, D. and Miikkulainen, R. (1998). Forming Neural Networks Through Efficient and Adaptive Coevolution. *Evolutionary Computation*, 5(4):373–399.

Moriarty, D., Schultz, A., and Grefenstette, J. (1999). Evolutionary Algorithms for Reinforcement Learning. *Journal of Artificial Intelligence Research*, 11:199–229.

Obayashi, S., Sasaki, D., Takeguchi, Y., and Hirose, N. (2000). Multiobjective evolutionary computation for supersonic wing-shape optimization. *IEEE Transactions on Evolutionary Computation*, 4(2):182–187.

Oh, S. and Lee, Y. (1995). Sensitivity Analysis of Single Hidden-Layer Neural Networks. *IEEE Transactions on Neural Networks*, 6(4):1005–1007.

Oviatt, S. and Cohen, P. (2000). Multimodal interfaces that process what comes naturally. *Communications of the ACM*, 43(3):43–53.

Paek, T. (2001). Empirical methods for evaluating dialog systems. In *2nd SIGdial Workshop on Discourse and Dialog*, Aalborg, Denmark.

Paek, T. (2006). Reinforcement learning for spoken dialogue systems: Comparing strengths and weaknesses for practical deployment. In *Interspeech Workshop on "Dialogue on Dialogues" - Multidisciplinary Evaluation of Advanced Speech-based Interactive Systems*, Pittsburgh, USA.

Pietquin, O. and Beaufort, R. (2005). Comparing ASR Modeling Methods for Spoken Dialogue Simulation and Optimal Strategy Learning. In *9th European Conference on Speech Communication and Technology (INTERSPEECH)*, Lisbon, Portugal.

Pietquin, O. and Dutoit, T. (2006). A Probabilistic Framework for Dialog Simulation and Optimal Strategy Learning. *IEEE Transactions on Audio, Speech and Language Processing*, 14(2):589–599.

Polifroni, J. and Walker, M. (2006). Learning database content for spoken dialogue system design. In *5th International Conference on Language Resources and Evaluation (LREC)*, Genoa, Italy.

Pon-Barry, H., Clark, B., Schultz, K., Bratt, E., and Peters, S. (2004). Advantages of spoken language in dialogue-based tutoring systems. In *7th International Conference on Intelligent Tutoring Systems*, pages 390–400, Maceio, Brazil.

Ramakrishnan, N., Capra, R., and Perez-Quinones, M. (2002). Mixed-Initiative Interaction = Mixed Computation. *ACM Sigplan Notices*, 37(3):119–130.

Raymond, C., Esteve, Y., Bechet, F., De Mori, R., and Damnati, G. (2003). Belief confirmation in spoken dialog systems using confidence measures. In *IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*, U.S. Virgin Islands.

Rechenberg, I. (1964). Cybernetic solution path of an experimental problem. Technical report, Library Translation 1122, Royal Aircraft Establishment, Farnborough, UK, 1965.

Reiser, V. and Lemon, O. (2006). Using machine learning to explore human multimodal clarification strategies. In *Proceedings of the ACL*, Sydney, Australia.

Rosenstein, M. and Barto, A. (2001). Robot Weightlifting By Direct Policy Search. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 839–846, Seattle, USA.

Roy, N., Pineau, J., and Thrun, S. (2000). Spoken Dialogue Management Using Probabilistic Reasoning. In *38th Annual Meeting of the Association for Computational Linguistics (ACL)*, Hong Kong, China.

Salomon, R. (1996). The influence of different coding schemes on the computational complexity of genetic algorithms in function optimization. In *Parallel Problem Solving from Nature – PPSN IV*, Berlin, Germany.

Schatzmann, J., Weilhammer, K., Stuttle, M., and Young, S. (2006). A survey of statistical user simulation techniques for reinforcement learning of dialogue management strategies. *Knowledge Engineering Review*, 21(2):97–126.

Scheffler, K. and Young, S. (2000). Probabilistic simulation of human-machine dialogues. In *International Conference on Acoustics, Speech and Signal Processing*, pages 1217–1220, Istanbul, Turkey.

Scheffler, K. and Young, S. (2002). Automatic learning of dialogue strategy using dialogue simulation and reinforcement learning. In *Human Language Technology Conference (HLT)*, San Diego, USA.

Schmidhuber, J. (2000). Evolutionary computation vs reinforcement learning. In *3rd Asia-Pacific Conference on Simulated Evolution and Learning (SEAL)*, Nagoya, Japan.

Schuurmans, D. and Schaeffer, J. (1989). Representational difficulties with classifier systems. In *3rd International Conference on Genetic Algorithms*, pages 328–333, Fairfax, VA, USA.

Seneff, S. and Polifroni, J. (1996). A New Restuarant Guide Conversational System: Issues in Rapid Prototyping for Specialized Domains. In *4th International Conference on Spoken Language Processing (ICSLP)*, Philadelphia, USA.

Shriver, S. (2001). A unified design for human-machine voice interaction. In *ACM Conference on Human Factors in Computing Systems*, Seattle, USA.

Shute, V. (1995). Smart: Student Modeling Approach for Responsive Tutoring. *User Modeling and User-Adapted Interaction*, 5(1):1–44.

Singh, S. and Bertsekas, D. (1997). Reinforcement learning for dynamic channel allocation in cellular telephone systems. In *Advances in Neural Information Processing Systems*, Denver, USA.

Singh, S., Kearns, M., Litman, D., and Walker, M. (2000). Empirical Evaluation of a Reinforcement Learning Spoken Dialogue System. In *17th National Conference on Artificial Intelligence (AAAI)*, Austin, USA.

Singh, S., Litman, D., Kearns, M., and Walker, M. (2002). Optimizing dialogue management with reinforcement learning: Experiments with the NJFun system. *Journal of Artificial Intelligence Research*, 16:105–133.

Smith, R., Dike, B., Ravichandran, B., El-Fallah, A., and Mehra, R. (2000). The Fighter Aircraft LCS: A Case of Different LCS Goals and Techniques. In Lanzi, P. L., Stolzmann, W., and Wilson, S., editors, *Learning Classifier Systems. From Foundations to Applications*, Lecture Notes in Artificial Intelligence 1813, pages 283–300. Springer-Verlag.

Smith, S. (1983). Flexible learning of problem solving heuristics through adaptive search. In *8th International Joint Conference on Artificial Intelligence (IJCAI)*, Karlsruhe, Germany.

Sun (2007). Sun Java JDK 1.5. http://java.sun.com/j2se/1.5.0/docs/.

Sutton, R. and Barto, A. (1998). *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, USA.

Sutton, S., Cole, R., de Villiers, J., Schalkwyk, J., Vermeulen, P., Macon, M., Yan, Y., Kaiser, E., Rundle, B., Shobaki, K., Hosom, P., Kain, A., Wouters, J., Massaro, M., and Cohen, M. (1998). Universal Speech Tools: the CSLU Toolkit. In *5th International Conference on Spoken Language Processing (ICSLP)*, pages 3221–3224, Sydney, Australia.

Syswerda, G. (1989). Uniform Crossover in Genetic Algorithms. In *3rd International Conference on Genetic Algorithms*, Los Altos, CA, USA.

Tesauro, G. (1992). Practical issues in temporal difference learning. *Machine Learning*, 8:257–277.

Tesauro, G. (1995). Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(1):58–68.

Tetreault, J. and Litman, D. (2006). Using Reinforcement Learning to Build a Better Model of Dialogue State. In *11th Conference of the European Association for Computational Linguistics (EACL)*, Trento, Italy.

Toney, D., Moore, J., and Lemon, O. (2006a). Developing conversational interfaces with XCS. In *9th International Workshop on Learning Classifier Systems (IWLCS)*, Seattle, WA, USA.

Toney, D., Moore, J., and Lemon, O. (2006b). Evolving optimal inspectable strategies for spoken dialogue systems. In *Human Language Technology North American Chapter of the Association of Computational Linguistics (HLT-NAACL)*, New York, USA.

van Kuppevelt, J., Heid, U., and Kamp, H. (2000). Special issue on best practice in spoken language dialogue systems engineering. *Natural Language Engineering*, 6(3–4).

van Zanten, G. (1999). User modelling in adaptive dialogue management. In *6th European Conference on Speech Communication and Technology (EUROSPEECH)*, Budapest, Hungary.

Walker, M. (2000). An application of reinforcement learning to dialogue strategy selection in a spoken dialogue system for email. *Journal of Artificial Intelligence Research*, 12:387–416.

Walker, M., Aberdeen, J., Boland, J., Bratt, E., Garofolo, J., Hirschman, L., Le, A., Lee, S., Narayanan, S., Papineni, K., Pellom, B., Poliforni, J., Potamianos, A., Prabhu, P., Rudnicky, A., Sanders, G., Seneff, S., Stallard, D., and Whittaker, S. (2001). DARPA Communicator Dialog Travel Planning Systems: The June 2000 Data Collection. In *7th European Conference on Speech Communication and Technology (EUROSPEECH)*, Aalborg, Denmark.

Walker, M., Hirschman, L., and Aberdeen, J. (2000). Evaluation for DARPA COMMUNICATOR Spoken Dialogue Systems. In *2nd International Conference on Language Resources and Evaluation (LREC)*, Athens, Greece.

Walker, M., Litman, D., Kamm, C., and Abella, A. (1998). Evaluating spoken dialogue agents with PARADISE: Two case studies. *Computer Speech and Language*, 12:317–347.

Watanabe, T., Araki, M., and Doshita, S. (1998). Evaluating Dialogue Strategies under Communication Errors Using Computer-to-Computer Simulation. *IEICE Transactions on Information and Systems*, 81:1025–1033.

Watkins, C. (1989). *Learning from Delayed Rewards*. PhD thesis, Cambridge University.

Watkins, C. and Dayan, P. (1992). Q-learning. *Machine Learning*, 8:279–292.

Whitley, D., Dominic, S., Das, R., and Anderson, C. (1993). Genetic reinforcement learning for neurocontrol problems. *Machine Learning*, 13:259–284.

Wilcox, J. (1995). Organizational learning within a learning classifier system. Technical Report No. 95003 IlliGAL, University of Illinois at Urbana-Champaign.

Williams, J. (2006). *Partially Observable Markov Decision Processes for Spoken Dialogue Management*. PhD thesis, Cambridge University.

Williams, J., Poupart, P., and Young, S. (2005). Partially Obervable Markov Decision Processes with Continuous Observations for Dialogue Management. In *6th SIGdial Workshop on Discourse and Dialogue*, Lisbon, Portugal.

Williams, J. and Young, S. (2006). Partially Observable Markov Decision Processes for Spoken Dialog Systems. *Computer Speech and Language*, 21(2):393–422.

Wilson, S. (1995). Classifier fitness based on accuracy. *Evolutionary Computation*, 3(2):149–175.

Wilson, S. (1998). Generalization in the XCS classifier system. In *Annual Conference in Genetic Programming*, pages 665–674, Madison, WI, USA.

Wilson, S. (2000). State of XCS classifier system research. *Lecture Notes in Computer Science*, 1813:63–81.

Wilson, S. (2001). Compact rulesets from XCSI. In *4th International Workshop on Advances in Learning Classifier Systems*, pages 197–210, San Francisco, CA, USA.

Yamauchi, B. and Beer, R. (1993). Sequential behavior and learning in evolved dynamical neural networks. *Adaptive Behavior*, 2:219–246.

Yankelovich, N. (1996). How do users know what to say? *Interactions*, 3(6):32–43.

Zadrozny, W., Budzikowska, M., Chai, J., Kambhatla, N., Levesque, S., and Nicolov, N. (2000). Natural Language Dialogue for Personalized Interaction. *Communications of the ACM*, 43(8):116–120.

Zue, V. and Glass, J. (2000). Conversational interfaces: advances and challenges. *Proceedings of the IEEE*, 88(8):1166–1180.

Zue, V., Seneff, S., Glass, J., Polifroni, J., Pao, C., Hazen, T., and Hetherington, L. (2000). JUPITER: A Telephone-Based Conversational Interface for Weather Information. *IEEE Transactions on Speech and Audio Processing*, 8(1):85–96.