VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@mcs.vuw.ac.nz

An Intelligent Agent that Learns in Complex, Novel, Environments

Maciej Wojnar

Supervisor: Dr. Peter Andreae

Submitted in partial fulfilment of the requirements for
Bachelor of Science with Honours in Computer Science.

Abstract

This report describes the design and implementation of an intelligent agent that learns STRIPS-like rules in a complex, relational, simulated world. These rules describe the behaviour of the actions in the world and can be used to make predictions and plans. The agent can learn compact rules despite the presence of irrelevant information. It can learn rules for actions with hidden effects and uses a novel algorithm for learning the conditional effects of actions. The report presents an exploration strategy that enables the agent to focus its learning and to use rules it has already learned to help learn new rules.

Contents

Chapter 1

Introduction

Autonomous robotic intelligent agents hold great promise for the future. Science fiction dreams such as household robots, robotic assistants in rest homes, rescue robots, and military robots are becoming a reality. A lot of progress has been made in low-level areas of autonomous robotic systems such as vision and locomotion that are vital for any kind of robot to exist. However there has been less work in the equally (if not more) critical area of higher-level, abstract reasoning that any useful intelligent agent must do [4].

1.1 The Problem

1.1.1 To act in the world one must have a broad knowledge base

The real world is a fantastically complicated place and it is amazing that humans can deal with this complexity so well that they take it for granted. The domain in which humans exist is broad but shallow. People have a wide array of actions to choose from, that they don't plan deeply. People excel at such broad, shallow environments while most algorithms developed in artificial intelligence perform much better in deep but narrow domains such as chess and logistics.

In order for an intelligent agent to be able to effectively interact in the broad, complex, relational world that people inhabit, it must be equipped with an equally broad knowledge base that usually does not have to be very deep. It has to know how to perform each of the many possible actions, what the effects would be, and whether or not it should perform the action in the first place. However, a really useful, flexible, and general-purpose agent cannot be equipped with all of the knowledge that it could ever need, because there is just too much of it. Instead, it should have a basic knowledge base and then be able to learn everything else it would require to act effectively in any new domain, by interacting and experimenting with the environment.

1.1.2 Learning that knowledge base is difficult

Learning the preconditions and effects of actions just by experimenting with them is actually very challenging. One of the biggest difficulties is sifting through the enormous amount of irrelevant data to find the kernels of important information. For example, suppose an intelligent agent is trying to learn how to pick up objects, and it has already successfully picked up several objects in the past. Now it tries to pick up a table and it fails. The agent has to determine why it failed. Possible reasons could be the table was too big, the time was 1:04pm, there was a book on the table, the table was in the kitchen, the light was on, etc.

The daunting task facing the agent is to determine which of these reasons are plausible or relevant and which have nothing to do with the problem.

More interesting and difficult actions that the agent needs to learn are those that have conditional and hidden effects. One single type of action can have completely different effects. For example, suppose that the agent is clumsy. The effect of picking up a cup and picking up a delicate egg would be quite different as the agent would probably crush the egg. The agent should learn the rule that picking up fragile things results in them being crushed.

Hidden effects are even trickier to learn because their effects cannot be observed directly. For example, locking a door has no visible effect that the agent can see. This action, however, prevents the agent from opening the door in the future. To see the what effect locking the door had, the agent has to try to open the door. But it does not know that it should try opening the door in the first place! Discovering the hidden effects of an action is thus extremely difficult to do.

The space of actions that the agent has to explore is quite large. The agent's exploration of this space should therefore be constrained. If the agent just tries random actions it will take a long time before it understands how any of the actions behave. But a more conservative exploration strategy (such as focusing on only one action until the agent is sure it knows how it works) can be much more effective. The agent could use actions it is already confident about to help it experiment with actions it is not confident about, thus facilitating its learning process.

1.2 Limitations of Standard Learning Techniques

Reinforcement learning and concept learning algorithms (such as neural networks, decision trees, bayesian networks, support vector machines, etc.) have been quite successful in the past in narrow domains such as pattern recognition. These algorithms are extremely effective at learning single concepts or tasks very well. However, standard concept learning is not suited to relational, complex domains; especially where there are sequences of actions and effects.

1.2.1 It is unclear how to represent first-order predicates

The world that people inhabit is filled with structure, which is difficult, if not impossible, to encode with simple propositional logic. However, most concept learning algorithms can only represent and learn expressions in propositional logic. For example, a neural network can represent simple propositional variables as inputs and outputs. To represent the propositional expression *black* $\wedge$ *dog* it could simply activate the input nodes corresponding to black and dog. However first-order logic has not only variables, but structure as well. It is unclear how a neural network would represent the first-order logic expression *black(dog)* $\wedge$ *grey(cat)*. It cannot just activate the input nodes corresponding to black, dog, grey, and cat because this does not specify that black corresponds only to dog and grey corresponds only to cat. Another example would be *dog chasing cat* where once again the structure of the predicates plays a key role (it is the dog that is chasing the cat and not the other way around). It is very difficult to encode such concepts of structure into standard neural networks or other standard concept learning algorithms [6, 4].

Reinforcement learning addresses the issue of learning what actions to perform to achieve a task. It is therefore much closer to the goal of this project than concept learning techniques. However, it is also limited to propositional representations and it is limited to learning a single task or goal.

1.2.2 Usually an unreasonably large number of examples is required

Probabilistic and statistical algorithms such as Bayesian networks, neural networks, support vector machines, etc., almost always require a large amount of training data in order to classify anything accurately. But this does not seem reasonable in the domain that people learn in. A person can learn how to distinguish most objects after seeing one or two examples. Standard neural networks can require hundreds and sometimes thousands of examples before they can do so with equal accuracy. An intelligent agent that can cope in the real world must also be able to learn from a very small number of examples.

1.2.3 It is unclear how knowledge learned in one problem could be transferred to a different problem

Standard concept learning algorithms excel at learning single concepts. However, it is difficult (and sometimes impossible) to transfer concepts learned in one problem to facilitate the solution of new problems. For example, once a neural network is trained for a particular problem or task, its solution is encoded in a series of weights (whose meaning is often impossible to extract). It is still an open research problem of how one would go about extracting such a solution to a problem and using it to solve a similar problem in the future.

1.3 Goals of the Project

This project addresses the issue of how an autonomous agents can learn the broad knowledge base they need to interact effectively with complex, relational worlds. It describes a series of agents that acquire knowledge about the world by interacting with it. This knowledge is in the form of rules that describe the behaviour of the actions that the agents can perform. The agent learns about actions by experimenting with them and seeing how the world changes. The task of an agent is to learn the best possible rules. The quality of a rule is determined by how well it can predict the effects of actions. A benefit of representing the knowledge base in this way is that it is easy to evaluate how good each rule is and the rules the agent learns can be used to form plans.

The project explores the mechanisms and heuristics that an agent needs to effectively learn such rules.

1.4 Road Map for this Paper

Chapter 2 presents some background relevant to this project. Chapters 3 describe the simulated world, Crane World, in which all of my agents are situated. Chapters 4-6 present three agents that I have designed and implemented that address some issues of high-level reasoning in complex, relational worlds. Each successive agent is a more sophisticated extension of the previous one. The first agent addresses the issue of separating relevant from irrelevant information. The second agent addresses the issue of learning conditional effects as well as determining relevancy. The final agent addresses the issue of hidden effects as well as determining relevancy and learning conditional effects. Chapter 7 summarizes the contributions of the project, identifies the limitations of the agents, and describes areas of future research.

Chapter 2

Background

This chapter presents three areas relevant to this project and then outlines an agent that addresses similar issues to this project.

2.1 STRIPS Rules

Richard E. Fikes and Nils J. Nilsson developed the STRIPS (STanford Research Institute Problem Solver) language for expressing rules about actions that could be used to form plans [3]. This language is especially suited to expressing conditions and effects in propositional and first-order logic. STRIPS has become almost ubiquitous because it is both expressive enough to express a wide variety of problems and yet constrained enough that it can be efficiently manipulated.

Actions in STRIPS are defined by the preconditions that must hold before the action can be executed and the effects the action has when it is executed. The action of picking up a cup off of a table would be represented in the following way:

```
Action(PickUp(cup, from))
Precondition:  At(cup, from) ∧ HasRoom(Agent)
Effect:  ¬At(cup, from) ∧ In(cup, Agent) ∧ ¬HasRoom(Agent)
```

However, the STRIPS language is quite limited. It does not allow negations or disjunctions in the preconditions, and it has no way of expressing conditional effects. The rules that my agents learn are loosely based on STRIPS but they are much more expressive.

2.2 Vision Systems and Tracking Objects

David Chapman developed an intelligent agent, Sonja, that could play an arcade game (Amazon) as well as a competent beginner [2]. Sonja plays the game in real time and so it must use reactive rules in order to be able to respond quickly enough to the changes in the game. Sonja can also interpret commands written in natural language. Only Sonja's vision system, however, is relevant to this project. When Sonya plays the game, it does not concentrate on the entire image (which can be quite large and complicated). Instead, Sonya focuses on sub-parts of the image that it deems to be important. Sonya also use visual markers (or tags) to track a limited number of important objects in the world (e.g. an amulet, a doorway, a ghost, etc.). I have used this idea of tracking a small number of objects in the vision system of the agents that I have developed.

2.3 Version Spaces

Version Spaces are a very powerful method of learning a hypothesis from positive and negative examples. The standard way of learning a hypothesis is to keep track of all the hypotheses that include all of the positive examples and exclude all of the negative examples. Eventually, given enough examples, this method will either produce a small set of consistent hypotheses, or produce no hypotheses at all (if there are no possible hypotheses consistent with all of the examples).

Unfortunately for any given problem, the space of possible hypotheses is usually extremely large and it is simply not feasible to store every single hypothesis that is consistent. Version Spaces allow one to keep track of every consistent hypothesis without having to explicitly remember every single one.

In Version Spaces one first applies a partial ordering on the hypotheses. Once the hypotheses have been partially ordered, in order to represent a large number of hypotheses it is enough to remember the bounding hypotheses: the "smallest" hypothesis in the group and the "largest" hypothesis. All the hypotheses in between are represented implicitly. Note that since the ordering is only partial, there may be several hypotheses that are "largest" and all of them have to be remembered.

The usual ordering that is applied to the space of hypotheses is the generality of each hypothesis. The more examples a hypothesis includes, the more general it is. To keep track of all of the consistent hypotheses one just has to keep track of the most specific consistent hypothesis and the most general consistent hypothesis. Every time that one sees a positive example, the most specific consistent hypothesis (also known as the most specific generalization, MSG) has to be generalized (just enough) to include the new positive example. Every time that one sees a negative example, the most general consistent hypothesis (also known as the most general generalization, MGG) has to be specialized (just enough) to exclude the new negative example. This process continues until either some solution set is found, or the Version Space collapses (i.e. there are no consistent hypotheses).

Unfortunately Version Spaces have a major flaw. It can take exponential time for a Version Space to converge to a solution set of hypotheses or to collapse showing that there is no solution. This means that they are not appropriate for problems that have a very large hypothesis space.

Version Spaces were invented by Tom M. Mitchell, and he and Ryszard S. Michalski and Jaime G. Carbonell used them to develop an intelligent agent that could learn what rules to apply to integration problems [5].

I have used concepts, such as the MSG and MGG, from Version Spaces when I developed the preconditions of the rules that my agents learn. The hypothesis space that the agents search is the space of possible rules.

2.4 An agent that learns rules in a relational world

LIVE, developed by Wei-Min Shen integrates activities such as perception, action, experimentation, learning, exploration, etc., all critical for intelligent agents that learn about their environment. This ambitious system was tested in a few simple domains with encouraging results. However, I believe that these domains were too simple to be used as reliable evidence for the effectiveness of the system.

The system (or the agent using it) interacts with its environment via triples of the form (BeforeState, Action, AfterState). It also has a model of the world in the form of rules stating what is expected to happen when each action is executed in various states.

If the LIVE system has no rule associated with an action it just executed, then it creates a new rule. It first looks at the state before and after the action. Each state consists of a list of facts (or percepts) about the world (e.g. On(Ball, Table)). It divides the percepts into three groups: the "vanished" group are all the percepts that were in the before state but not in the after state, the "merged" group are those in the after state but not in the before state, and those that are in both the before and after states. The LIVE system sets the precondition of the rule to be vanished percepts and the negation of merged percepts. It sets the postcondition to be the merged percepts and the negation of the vanished percepts.

Just before executing an action, the system makes a prediction about the result of the action. It then compares the prediction and the actual result of the action. The LIVE system only learns from prediction failures. After a failure, the system modifies its model to account for it, otherwise it does nothing. If there is a failure then the rule responsible is split into two rules. The rules represent the two predictions (the surprising one and the original one) and their preconditions are modified so that the state that caused the surprising result is added to one rule and removed from the other. If the rule that failed has already been split then one of its sub-rules is generalized and the other is specialized to account for the failure.

The system explores by choosing actions whose outcome is uncertain, in order to improve the system's model of the world. Exploration is quite difficult because the search space is usually enormous. To constrain its exploration, the LIVE system uses three heuristics:

- It explores actions it thinks will lead to the goal
- It investigates actions that apparently have no effect
- Sometimes it randomly picks an unexplored action to explore

The system experiments by making a prediction sequence whose final prediction is expected to fail. If the final prediction does indeed produce a prediction failure, then this is very useful for revising the system's model of the world.

The system is also capable of discovering new relations and terms. For example, in a see-saw world, it could discover torque by multiplying weight and distance. The LIVE system uses a brute-force approach to try to discover important relationships.

The LIVE system starts running by attempting to form a plan to achieve some goal. If it cannot achieve the goal because it has not formed enough rules about the actions, then it starts exploring. If it cannot achieve the goal because its predictions are wrong, then it experiments with the predictions that were wrong by forming plans that it thinks will make the predictions fail again (in order to adjust its model of the world to explain the failure).

The general idea of planning and then either doing exploring or experimenting, is a very interesting one that I believe is worth investigating. I also think that splitting rules in order to handle conditional effects is the correct approach.

However, the LIVE system has a number of serious limitations:

- The (rather trivial) example that the author uses to test the LIVE system is the famous Tower of Hanoi problem. The task of the agent is to solve a tower of depth three without any knowledge about the rules of the world (eg. you cannot put down a larger disk on smaller disk, you cannot pick something up if you are already holding something, etc). There are only two actions to be learned (pick up and put down) and only six objects in the world (the three disks and three plates) that lack any sort of detailed description. The other domains in which the LIVE system was tested were also very simple.

- The system only ever learns from failures. It cannot get better from just seeing positive examples.

- The LIVE system discards too much potentially valuable information when constructing rules. It ignores percepts that are present in both the before and after states. For example: when picking up a ball it would completely ignore its weight since this property is not changed by the action. The other consequence of this is that if two different objects in the world had the same attribute, the LIVE system would not see any difference between the two percepts. For example if there was a dirty ball with the percept Clean(No) and a dirty chair with the percept Clean(No) and the agent did the action Wash(ball), then the LIVE system would see that a percept has changed from Clean(No) to Clean(Yes) but it could not tell which object had become clean. The LIVE system would not work very well in domains with a lot of irrelevant information.

- When an action produces a surprising effect, its rule is split into two new sub-rules to handle both the original and the new effect. However, a rule is only every split once. This means that the system cannot handle actions that may have three or more possible conditional effects.

- The system cannot learn hidden effects. Its mechanism for discovering new relations relies on an exponential algorithm that simply does not scale.

The fundamental flaw of the LIVE learning system presented in this paper is that it will not scale to more interesting and complex worlds.

Chapter 3

Crane World and Learning Agents

3.1 General Description

The simulated world that I am using for this project is called "Crane World". It was designed and implemented by David Gilligan. Although simplistic, this world is specifically designed to model the breadth of the real world. This means that it has a lot of objects each having a lot of detail, and a lot of actions that can be performed. There are usually between 20 and 80 possible actions in each state. For example: Pick up cup, Pick Up book, Open door, Lock door, Operate sink, Put Down sugar On table, Close box, etc. Note that most of the detail in the world is completely irrelevant to any particular action.

There is also an intelligent agent that exists in this world. It receives information about the state of the world, and it can perform actions to alter the state of the world in some way. The agent's ultimate goal is to be able to predict the outcome of every action perfectly.

3.1.1 The state of the world consists of a list of objects and their descriptions

Crane World has a large number of objects each with a large number of attributes. The objects can also have relationships amongst each other (e.g. a cup may be related to a table by being ON it and having the table SUPPORT it). An example of an object in Crane World would be:

```
isa[Table], Colour[brown], In[Kitchen], Supports[Cup, Plate]
```

This description (the object's descriptor) represents a brown table that is inside the kitchen and has a cup and plate on top of it. Note that these relationships only express what object types are related to a given object not that actual identities of the objects. In the above example, the descriptor does not convey any information about which cup and plate are actually on the table.

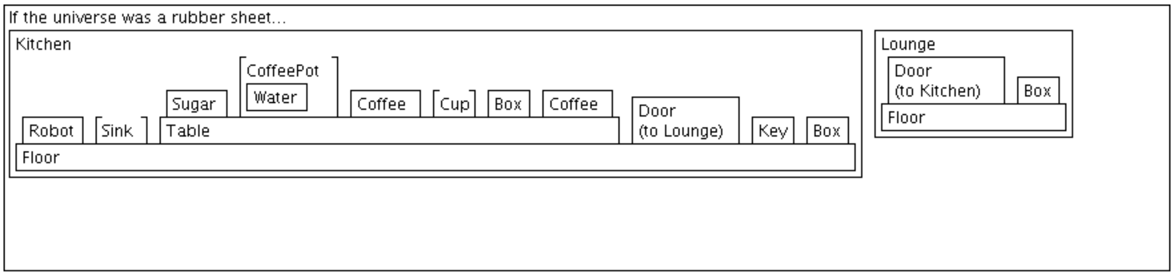

Figure 3.1: The GUI of Crane World

Every object in Crane World has a descriptor (like the one above) describing it. It consists of the object's attributes with their corresponding values. Note that most of the information in the descriptor is usually irrelevant, for example the colour of the table is irrelevant when one is deciding whether or not the table can be picked up. The entire state of the world can be described by simply listing out all of the objects and their descriptors.

Within Crane World there is a special object called the "Robot". It represents the "physical body" of the intelligent agent that exists in this world.

3.1.2 The agent can perform actions to modify the state of the world

The agent can affect the world by performing actions in it. These actions involve either one or two objects in the world. These objects that are directly related to the action are called the arguments of the action. The arguments take part in the action in some way. For example, PutOn(A, B) takes two objects as arguments (A and B). Object A is the thing that is being put and object B is the thing on which it is put. Note that an action is structured in a similar way to English verbs. In the example above, A would be the direct object and B would be the indirect object.

The action modifies the world by changing objects in the world (adding or removing attributes or properties) or by adding or deleting objects. For example, the action PutOn(cup, table) would change three objects in the world: the cup, the table, and the robot. The state:

```
isa[Cup], In[Robot], ...
isa[Table], ...
isa[Robot], Contains[Cup], ...
...
```

would change to:

```
isa[Cup], On[Table], ...
isa[Table], Supports[Cup], ...
isa[Robot], ...
...
```

(Note that I have used ... to indicate that the agent would actually receive a lot more irrelevant information, that I have omitted here for brevity).

3.2 The Percept-Action-Percept cycle

The agent cycles through two main states: a Percept state, and an Action state. In a Percept state, the agent receives information about the current state of the world. In an Action state the agent receives a (long) list of possible actions it can perform, and it selects one. Crane World executes the agent's action by modifying the world in an appropriate way. The agent then enters a new Percept state.

The agent cycles through states of Percept1, Action1, Percept2, Action2, ..., PerceptN, ActionN, ... Each percept (except for the first one) is an effect of the previous action and it is the precondition for the next action. In general, the agent learns rules about actions by examining (Before Percept, Action, After Percept) triples.

3.2.1 In the Percept phase the agent receives information about the current state of the world

The agent receives information about the state of the world in the form of percepts. A percept is a list of all the object descriptors in the world that the agent can currently sense. The agent can sense only those objects that are in the same room that are not within any closed container. Note that objects can have attributes that the agent cannot sense directly. For example, the robot cannot tell whether a door is locked or not by examining its descriptor directly.

3.2.2 In the Action phase the agent receives a list of "plausible" actions and chooses one

The goal of this project is for the agent to learn complex rules, and the conditional and hidden effects of the possible actions. I assume that the agent already knows how to "physically" perform each action. So it is reasonable to assume that initially the agent knows the most basic things about each type of action. For example, it would know that it does not make sense to try to Pick Up the floor or to Open a fork.

In the Action phase the agent chooses only such plausible actions (plausible actions are predefined by Crane World). Note that just because an action is plausible is no guarantee that it will succeed or have the expected effects. Opening a door is a plausible action, but if the door is locked, then the effect of the action will be unexpected (the door will not open).

3.3 Vision System

3.3.1 The agent can focus on particular objects in the world

The basic way that the agent perceives the world is by getting a list of descriptors of the objects around it. I refer to this as its "Vision System". In the original Crane World implementation the agent is not given the identity of any of the objects, all it gets is each objects description. However, this severely limits the agent and forces it solve unrealistic problems.

For example, if there were three identical cups sitting on different tables and the agent wanted to pick up one of them, then the agent should be able to predict which table will no longer support a cup. If the percepts only contain descriptions of objects, then this can't be done because the agent can't distinguish the three cups in the percept.

The initial percept with no tagging would be:

```
isa[table], supports[cup], colour[green]
isa[table], supports[cup], colour[blue]
isa[table], supports[cup], colour[red]
isa[cup], on[table], colour[black]
isa[cup], on[table], colour[white]
isa[cup], on[table], colour[grey]

Action \texttt{PickUp cup}
```

However, I believe that it is reasonable to give an agent a focus of attention and allow it to track the identity of a limited number of objects in the environment (after all, people can focus on a small number of objects at once). With such a focus of attention the vision system would track a small number of objects and identify them even when they are changing during an action.

I therefore modified the Vision System to allow the agent to focus on up to two objects at a time. Whenever the agent performs an action, it focuses on the objects in the argument of the action. So if the agent performs a PutDown(Book, Floor), it focuses on both the book and the floor. Objects that the agent is focused on will have tags associated with them. These tags are only there for the duration of the action, after which the agent is no longer focused on the objects, and they becomes like every other object in the percept.

With this tagging, the before and after percepts above would be augmented.

The initial percept with tagging would be:

```
isa[table], supports[cup], colour[green]
isa[table], supports[cup#tag0#], colour[blue]
isa[table], supports[cup], colour[red]
isa[cup], on[table], colour[black]
isa[cup], on[table], colour[white]
#tag0# isa[cup], on[table], colour[grey]
isa[robot]

Action \texttt{PickUp cup}
```

The resulting percept would be:

```
isa[table], supports[cup], colour[green]
isa[table], colour[blue]
isa[table], supports[cup], colour[red]
isa[cup], on[table], colour[black]
isa[cup], on[table], colour[white]
#tag0# isa[cup], in[robot], colour[grey]
isa[robot], contains[cup#tag0#]
```

Tagging also automatically specifies what role various objects play in the action. In the above example, the blue table supported the object being picked up. The notion of what role objects play is very useful when doing object matching (see 4.4.1).

Chapter 4

Learning Agent 1: Dealing with Relevancy

4.1 Overview

I have implemented an Exploratory Agent for Rule Learning (EARL-1) that learns STRIPS-like rules (see 2.1) that describe the behaviour of the various actions. It learns by interacting with its environment.

When EARL-1 is in the Action state (see the Percept-Action-Percept cycle in 3.2), it selects an action from the list of possible actions, based up its exploration strategy. The purpose of the exploration strategy is to facilitate EARL-1's rule learning (see 4.5).

After choosing the action but before executing it, EARL-1 uses its rules to make a prediction as to what it thinks the result will be. EARL-1 makes these predictions in order to determine how correct its rules are by comparing its prediction to the actual result.

After performing the action, EARL-1 enters the Percept state, it uses the previous percept, the action, and the new percept to either learn or refine a rule associated with that action. EARL-1 learns a new rule for an action type that it has never seen before. If it already has a stored rule for the action type, then it refines the rule by either generalizing it if it sees a positive example (to include the new positive example) or specializing it if it sees a negative example/failure (to exclude the negative example).

EARL-1 assumes that an action can have only two kinds of effects: a normal effect or a failure. It assumes that if an action has no effect (that the agent can see) then it failed. This is almost always true. However, sometimes an action can succeed and have no visible effect (e.g. unlocking a door), and some failures may have visible effects (e.g. picking up an egg and breaking it, would probably be considered a failure even though there are visible effects of the action, namely a mess on the floor). Chapter 5 addresses the issue of such conditional effects.

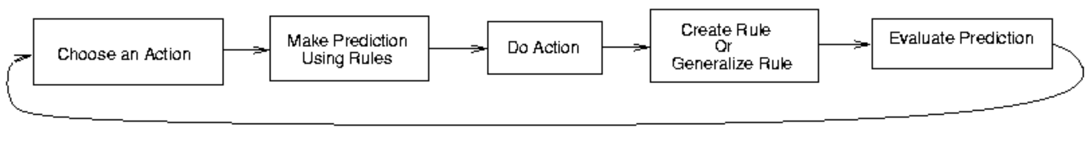

Figure 4.1: The rule learning cycle of the agent

A typical Percept (see Chapter 3) that the agent could receive is:

```
Perception =
[(Clean, [yes]), (Colour, [Clear]), (In, [Sink]), (isa, [Water])]

[(Clean, [yes]), (Colour, [white]), (In, [Kitchen]),
(On, [Table]), (isa, [Sugar])]

[(Clean, [yes]), (Colour, [Orange]), (In, [Kitchen]),
(On, [Cup]), (Open, [No]), (isa, [Box])]

[(Clean,[Yes]), (Colour, [steel]), (Contains, [Water]),
(In, [Kitchen]), (On, [Table]), (Open, [Yes]), (isa, [CoffeePot])]

(Clean, [yes]), (Colour, [brown]), (In, [Kitchen]), (On, [Floor]),
(Supports, [Coffee, CoffeePot, Sugar, Cup]), (isa, [Table])]

[(Clean, [yes]),(Colour, [brown]), (In, [Kitchen]), (On, [Table]),
(isa, [Coffee])]

[(Clean, [yes]), (Colour, [Orange]), (In, [Kitchen]),
(On, [Floor]), (Open, [No]), (isa, [Box])]

[(Clean, [no]), (Colour, [Grey]), (In, [Kitchen]),
(Supports, [Table, Sink, Key, Door, Box, Robot]), (isa, [Floor])]

[(Clean, [yes]), (Colour, [Brown]), (In, [Kitchen]),
(On, [Floor]), (Open, [No]), (isa, [Door])]

[(In, [Kitchen]), (On, [Floor]), (isa, [Robot])]

[(Clean, [yes]), (Colour, [Clear]), (In, [CoffeePot]),
(isa, [Water])]

[(Clean, [No]), (Colour, [white]), (In, [Kitchen]), (On, [Table]),
(Open, [Yes]), (Supports, [Box]), (isa, [Cup])]

[(Clean, [yes]), (Colour, [StainlessSteel]), (Contains, [Water]),
(In, [Kitchen]), (On, [Floor]), (Open, [Yes]), (isa, [Sink])]

[(Contains, [Table, Coffee, Sink, Key, Door, CoffeePot, Sugar,
Floor, Cup, Box, Box, Robot]), (isa, [Kitchen])]

[(Clean, [yes]), (Colour, [Iron]), (In, [Kitchen]),(On, [Floor]),
(isa, [Key])]
```

For any given action, only a very small part of the percept is actually relevant. If the agent wanted to open the door then only the highlighted part of the percept above would actually be important.

When EARL-1 generates a new rule it has to identify and discard irrelevant objects. It also has to identify what the effects of the action were. When generalizing rules it has to match the objects in the new rule to their corresponding objects in the existing rule, and identify the differences.

4.2 Rules Learned

Each rule that EARL-1 learns consists of a precondition and a postcondition. The precondition specifies what should be true about the inputted percept (the state of the world) in order for the rule to apply; the postcondition specifies how the state of the world will be affected by the action if the rule applies.

4.2.1 The precondition of each rule is a Most-Specific-Generalization

The precondition of each rule consists of a (generalized) list of objects. For a given percept, the precondition holds if and only if each of its objects can be satisfied by an object in the percept. An object in the precondition is satisfied by an object in the percept if the object in the precondition has all of the same attributes (or more) as the object in the percept and the values of the attributes in the precondition subsume the corresponding values in the percept object.

EARL-1 assumes that if an attribute has been generalized to contain more than one value, then it can subsume any value. This is noted by an (Anything) after the attribute name. So if EARL-1 picks up a cup and then a book then it generalizes its pick up rule to state that the type of the object picked up is irrelevant (i.e. the "isa" attribute does not constrain the rule in any way).

This corresponds to a (1,Many) counting system: an attribute that has been generalized to subsume "Anything" as its value will subsume every other attribute of the same type. This is not a very good generalization to make in the real world because quite often it is not true. Just because a person can pick up a table supporting a cup or a table supporting a plate does not mean that what the table supports is unimportant. They would not be able to pick up a table supporting a car. However, this (1, Many) counting system is quite useful in Crane World as it speeds up the learning process.

Even when an attribute has been generalized to be "Anything", the attribute still contains a list of all specific attribute values. These are used in constructing "unless" clauses (see 4.4.3).

For example the generalized object in the precondition:

```
[isa[chair], supports(Anything)[cup, plate, sugar],
colour(Anything)[white, black, green, blue],  heavy[yes]]
```

would match with this percept object

```
isa[chair], colour[yellow], heavy[yes]
```

In essence the precondition is a most specific generalization. Initially, when a rule is first created, it is specific to one example (it is specific to all of the objects that have not been ignored as irrelevant). As EARL-1 sees more and more positive examples, it generalizes the precondition just enough to include them.

4.2.2 The postcondition describes how objects are changed by the action

Every rule also has effects, or postconditions. The postcondition, like the precondition, consists of a list of objects. It describes how each object in the precondition is changed by the action. These changes (indexed by the object's type, see 4.4.1) can be the addition or deletion of individual values of an attribute of an object (e.g. +Clean[Yes], -Supports[Ball], +Colour[Yellow], etc.), the deletion of an entire attribute (e.g. "-In" signifies that after the action, the object will no longer be in anything), or the deletion of the entire object. The postcondition also specifies any objects that are added as a result of the action.

4.2.3 An "unless" clause deals with failures

Nearly every action available to the agent can fail. For example, opening a locked door will fail (there will be no effect). To be able to create rules that can properly predict failures, it associates an "unless" clause with each rule that describes the conditions under which the action will fail. However, unlike the precondition, the unless clause is a disjunction and is a set of maximally general generalizations, rather than a maximally specific generalization.

The unless clause is structured in a similar way to the precondition. There is a sub-clause for each object in the precondition and a list of "facts" (things that were true about the object when the action failed) for each sub-clause. The unless clause holds if at least one of its sub-clauses can be satisfied by an object in the percept. An object satisfies the unless sub-clause if it has at least one attribute value pair in common with it. For example, the following unless clause (for the PickUp rule) specifies that a failure will occur if the object being picked up is a table, or if it supports a cup or plate or if the object supporting the object being picked up is a floor or if it is green:

```
Unless
ARGO: isa[table], supports[cup], supports[plate]
OBJ1: isa[floor], colour[green]
```

Due to the way that EARL-1 learns the precondition and unless clauses, the two are disjoint and so a percept can never satisfy both the precondition and the unless clause at the same time. When predicting, if the unless clause holds, EARL-1 predicts that there will be a failure (the state of the world will not change).

Under certain preconditions some actions can have other, unusual effects. However, EARL-1 only considers failures.

4.3 Learning New Rules

4.3.1 The agent matches objects in the before percept to those in the after percept

1

EARL-1 learns rules by examining the (before percept, action, after percept) triple. The before and after percepts are detailed descriptions of all the visible objects in the world. In order to form any kind of rule for the action that just occurred, EARL-1 must first identify which objects have changed (or been added or deleted) and how. To determine this, each object in the before percept must be matched to an object in the after percept. Only then can EARL-1 identify what the precondition and effects of the action were.

For each object in the before percept, EARL-1 finds a best matching object in the after percept. An object is a vector of attributes and their values, so a best matching object is

[1] This basic object matching procedure was provided with Crane World

one that has the most elements in common. If the number of differences between the object and its best match is above a certain threshold, then the agent assumes that the object has been deleted (if the best match is very different, then the object does not match any objects in the postcondition so a deletion had to have occurred). After each object in the before percept has been matched or noted as deleted, then any remaining unmatched objects in the postcondition are noted as being added.

This matching algorithm works quite well in cases where the before and after percepts are quite similar. However when the before and after percepts differ significantly (for example when moving to another room), then this algorithm can make mistakes. To improve its performance, I added the heuristic that two objects whose "isa" attributes are different cannot be matched (meaning one object cannot transmute into another type of object).

4.3.2 The agent uses a simple heuristic to eliminate irrelevant objects

Since most of the precondition percept is irrelevant, most of it should be discarded and not used to generate the rule. EARL-1 uses the heuristic that any objects that have not changed (their descriptions are the same before and after an action) are irrelevant and can be ignored. Though this is a very useful heuristic in general, it can sometimes eliminate an important object. For example, the effect of putting toast in the toaster is heavily influenced by the position of the dial (whether light, medium, or dark), but the dial does not change as a result of the action, so the heuristic would consider it irrelevant.

4.3.3 The agent calculates the preconditions and effects of the current action

The precondition of the current action consists of all of the objects in the before percept that were not eliminated by the heuristics.

To calculate the effects of the current action, EARL-1 compares each object before the action with the corresponding object after the action and works out what the differences are. It eliminates all the attributes that are shared (have the same name and the same values). Then it matches attributes of the same name but with different values and sees which values have been added and which have been deleted. It notes these changes as +a[v] if a value v has been added to attribute a and as -a[v] if it has been deleted. Finally it sees what attributes have no matching counterparts. Those attributes present in the object before the action but not present after, are noted as deleted, and those attributes present in the object after the action but not before, are noted as added. If an attribute a[v1, v2, v3] is added then this is noted as a set of value additions: +a[v1], +a[v2], and +a[v3]. All of these changes together are the effect of the action on that particular object. These effects are then associated with the corresponding object in the precondition. Note that EARL-1 records the addition and deletion of objects when it matches the objects in the before percept to those in the after percept.

4.4 Refining Stored Rules

4.4.1 The agent matches the objects in the new rule to the objects in the corresponding stored rule

Every time EARL-1 executes an action it creates a new rule for it (see 4.3.3). If it already has a stored rule for the same action type, then it generalizes the precondition of the stored rule with the new precondition and new effect it calculated in the previous step. In order to do this, EARL-1 must first match the objects in the newly calculated precondition to the objects in the precondition of the stored rule.

The arguments of the action are specified by the action itself (the action provides their identity directly), so there is no need to search for a match for them. Matching the rest of the objects is non-trivial. A best match approach is almost always wrong, because EARL-1 should match objects that play the same role in the action and not necessarily those that are the most similar. For example, if EARL-1 picks up a cup off of a saucer, and then picks up a book off of the floor, then it should match the saucer to the floor because they play the same role in the action (they supported the object being picked up), despite the fact that the saucer and floor are quite different from each other and may be more similar to other objects.

To determine the role objects play in the action, EARL-1 determines how they relate to objects in the arguments of the action. If two objects have the same tags (see 3.3) in the same attributes (in the precondition and in the postcondition), then these two objects match and they are generalized.

For example,

```
isa[floor], ..., supports[cup, book#tag0#, plate, ...]
```

and

```
isa[saucer], ..., supports[cup#tag0#]
```

would match because both objects supported the object being picked up (argument 0).

Most objects that need to be matched in this way (i.e. those that have not been eliminated by the heuristics for eliminating irrelevancy) are related directly in some way to at least one object in the action's arguments.

For objects that have no such direct relationship with the action's arguments (e.g. when turning on a light, the light has no direct relationship with the switch) EARL-1 matches them using a best match. In this case a best match is not necessarily bad. Objects that do not have such direct relationships tend to be constant (e.g. turning on switches usually turns on lights) and a best match works quite well.

4.4.2 The agent generalizes its stored rule with the new rule

During the process of generalizing a rule, once an object from the precondition of the new rule has been matched with an object in the precondition of the rule, the agent drops all of the attributes that are not common to both objects. If an attribute is present in only one object but not present at all in the other, then this attribute is clearly of no importance and should be removed from the rule. For example, if generalizing [isa(Ball), clean(yes), ..] and [isa[Cup], ...] the attribute "clean" is dropped because it does not occur in the second object.

EARL-1 then generalizes each object in the precondition by generalizing all of the attributes it has in common with its corresponding object in the before percept. For each attribute of an object in the rule's precondition, EARL-1 finds the corresponding attribute in the matched object. EARL-1 assumes that if an attribute has been generalized to contain more than one value, then it can contain any value (noted by an (Anything) after the attribute name). So if the agent picks up a cup and then a book, it generalizes its pick up rule to state that the type of the object is irrelevant (i.e. the "isa" contains Anything).

EARL-1 also extends the value list of the attribute of the object in the rule's precondition to contain all of the new values found in the corresponding attribute's value list. This is used when constructing the unless clause (see 4.4.3). An example of generalizing an object in the precondition would be the following:

Rule Precondition:

```
Isa[chair],
Supports(Anything)[cup, plate, sugar],
Colour[White],
Heavy[Yes]
```

Matched Object:

```
Isa[table],
Supports[tea, saucer],
Colour[Brown],
Clean[Yes]
```

The attributes in the rule's precondition would be generalized to:

New Precondition:

```
Isa(Anything)[chair, table],
Supports(Anything)[cup, plate, sugar, tea, saucer],
Colour(Anything)[Brown, White]
```

EARL-1 also generalizes the postconditions of the rule with the newly calculated effects of the action just executed. In general, postconditions are just collated together. So +in[kitchen] and +on[table] becomes (+in[kitchen], +on[table]), and -clean[yes] and -on[floor] becomes (-clean[yes], -on[floor]). However, two value deletions of the same attribute are generalized to be a single attribute deletion. For example, -on[table] and -on[floor] would be generalized to be -on. Although it may be better to wait for more examples before generalizing to an attribute subtraction, it is consistent with the (1, Many) counting system that EARL-1 uses.

The result of an action can also be the addition and/or deletion of entire objects. If only one object is added or deleted then EARL-1 notes down its descriptor. In the future, when predicting, it will predict that that exact descriptor will be added to or deleted from the percept representing the state of the world. However, when generalizing, if the new rule has added or deleted objects in its postcondition that are not equal to the added/deleted objects in the old rule, then EARL-1 generalizes the postcondition by adding "add many" or "delete many" to it. The meaning of "add many" (or "delete many" for deleted objects) is that the rule predicts that some objects will be added (or deleted for "delete many") but it is not expressive enough to specify which.

Note that it does not make sense to generalize the object to be added. It does make sense to generalize objects to be deleted, but not if they are "sufficiently" different. If different objects are added or deleted then this should really be handled with conditional effects (see Chapter 5).

4.4.3 If the action failed, then the agent constructs an "unless" clause

Every time an action fails, EARL-1 extracts every separate fact about each object in the before percept (that was not eliminated as irrelevant by the heuristics) and adds it to a list of possible reasons why the action failed. EARL-1 then removes any facts that are also in the precondition, because if a fact appears in the precondition then it occurred at least once, before a successful action, so it cannot be the cause of failure. For example, given the following precondition:

```
Precondition
ARG0: isa[chair], in[kitchen], clean[yes]
...
```

and the following object in the before percept:

```
ARG0: isa[table], in[ktichen], supports[cup, plate, saucer]
```

EARL-1 would construct the following unless clause:

```
Unless
ARG0:  isa[table], in[kitchen], supports[cup], supports[plate],
supports[saucer]
```

Note that the fact "ARG0: in[ktichen]" could not have been the cause of failure because it already appears in the precondition.

After extracting this list of facts, EARL-1 adds it to the corresponding list in the unless clause (in the sub-clause corresponding to the same object). This is a method of generalizing the unless clause to include more cases of when the action failed. To make sure that each unless sub-clause is disjoint from the corresponding object in the precondition, EARL-1 updates this list whenever the precondition is modified. During this update it scans through the list of facts in the unless clause and removes any facts that it has in common with the corresponding object in the precondition. This is a method of specializing the unless clause to exclude cases when the action succeeded.

Keeping a list of attribute values in the precondition for attributes that have been generalized to be "Anything", allows EARL-1 to express conditions such as: "The object being picked up can be any colour except red":

```
Precondition
ARGO: colour(Anything)[blue, green, yellow], ...
...

Unless
ARG0: colour[red]
```

Note that EARL-1 assumes that each fact in the unless clause is independent from every other fact in the unless clause. However this is not always the case. For example a person can pick up heavy things and they can pick up things when they're tired, but they can't pick up a thing if it is heavy AND they are tired. In this example the algorithm would be unable to find a correct rule. Chapter 5 addresses the issue of elements being dependent on each other.

4.5 Exploration Strategy

The exploration strategy guides EARL-1's selection of actions. It helps EARL-1 avoid doing a random search by having it focus on a particular type of action. It also helps EARL-1 avoid exploring rules it has already learned successfully and it stops EARL-1 from trying to learn actions that it is hopelessly confused about.

4.5.1 The agent checks its prediction against the new percept

EARL-1 has a confidence number associated with every rule governed by how well the rule predicts the effects of he action. Every time its prediction is correct, EARL-1 increases its confidence about that action by one. If the prediction does not match the new state, then it halves its confidence factor. So EARL-1 becomes confident gradually (linearly), but loses its confidence quickly (exponentially) if its predictions fail.

EARL-1 also has a confusion number for every rule. Every time its prediction is correct it resets its confusion number to 0. Every time the prediction fails, it increments its confusion number.

After every action EARL-1 compares the prediction it made for the last action with the new percept (see section 4.5.3). The prediction is correct if and only if it matches the new percept exactly. However, when the rule predicts that multiple objects will be added/deleted (by having "add many"/"delete many" in the postcondition), then an exact match is required only of the objects that have not been added/deleted. If the postcondition has "add many", then this part of the prediction will succeed as long as some objects have been added; if the postcondition has "delete many", then this part of the prediction will succeed as long as some objects have been deleted.

4.5.2 The agent selects its next action using an exploration strategy

EARL-1 has a focus of attention on action types. Whenever EARL-1 is in the Action state, it must select an action from the list of possible actions. It does so by selecting an action at random whose type (e.g. Pick Up) matches its focus. In the initial, state EARL-1 picks an action type at random and focuses on it.

EARL-1 switches its focus to a new action type if there is no possible action with the old action type to choose or when it becomes bored with the old action type, either because it has already learned everything there is to learn about that action or because it cannot make any headway and has become bored with constantly making bad predictions.

When the confidence level for a rule goes above a certain threshold (5), then EARL-1 becomes "confident" about the rule and switches its focus to a new action type. When the confusion level for a rule goes above a certain threshold (3), then it becomes "confused" with the rule and also switches its focus to a new action type. The confidence/confusion with rules is an example of constraining exploration using novelty [1].

Whenever EARL-1 switches its focus of attention to a new action type, it focuses on an action type that it is not already confused or confident about, if there are any. If there aren't any, then EARL-1 tries each of the actions that it is confused about until it becomes confused again. If EARL-1 has not made any progress (the number of actions that it is confused about is exactly the same), then it asks the teacher (human user) for help. The teacher then selects which actions EARL-1 should perform, but the rule learning algorithm still runs in the background. The teacher can thus guide EARL-1 by giving it a positive example. In order to form any kind of rule about an action, EARL-1 needs at least one positive example (to get the matching to work correctly). The most common reason that it becomes confused about a rule, is that it cannot get a single positive example of the action succeeding. If the teacher provides one, then this is often enough to get EARL-1 to learn the correct rule by itself. Note that when EARL-1 asks for the teacher's help, the teacher can refuse. In that case it just tries the action types that it is confused about, and if it cannot make any progress then it asks for help again.

Eventually EARL-1 will reach a point where it is either confident or confused about every action type. If it is confident about everything and confused about nothing, then it prints this out and stops trying actions.

4.5.3 The agent makes a prediction

After selecting an action but before executing it, EARL-1 makes a prediction by creating a copy of the current percept and altering it according to the postcondition of the rule corresponding to the action (if there is no rule, then it makes no prediction).

The changes stored in the postcondition are stored according to object type (the role that the object played in the action). So, to make a predicition, EARL-1 must match every object in the postcondition with an object in the percept. This is more challenging than the matching done during rule learning because EARL-1 does not have as much information available. During rule learning EARL-1 has both the before state and the after state and so it can match some objects according to the way they are changed by the action. For example in the Pick Up rule, Robot object can be matched because it ends up containing the object being picked up (note that this occurs after the action is executed). When predicting, however, EARL-1 can only match the objects based on their state before the action is executed, as it does not know what their state will be after the action is executed because this is what it is trying to predict!

Once again the identity of the objects in the arguments of the action are specified directly by the action so no matching is necessary. EARL-1 first matches objects that it can match perfectly because their roles depend on tags that occur solely in the preconditions. For example, if EARL-1 wants to pick up a ball from a table, then the role of the table in the action is that it supports the object being picked up, and this role can be determined completely from the precondition. EARL-1 then does a best match for all of the remaining objects in the postcondition that have tags in the postcondition. Note that despite the use of a best match, this is still quite effective because objects whose role in the action is determined after the action has been executed are usually constant across different actions. For example, in every Pick Up action, the Robot ends up containing the object being picked up.

Once EARL-1 has matched all the objects in the rule, it retrieves the corresponding postconditions and applies the changes stored there. Note that negative effects (e.g. -in) are performed before positive ones. This is to prevent the effects from cancelling each other out. For example, if +in[robot] were performed first, then -in would undo this effect.

If the postcondition also states that an object will be added or deleted, then EARL-1 adds the object to the percept or removes an object (if it matches exactly with the object to be deleted) from the percept. If multiple objects are to be added or deleted then EARL-1 simply adds "add many" or "delete many" respectively to the postcondition (see 4.5.1). EARL-1 then returns the selected action to Crane World, enters the Percept state, and starts the rule learning cycle again.

4.6 Example

This is an example of EARL-1 learning the pick up rule after picking up a cup, a book, and then attempting to pick up a table (and failing). The following is an example sequence of percepts and the resulting rules that EARL-1 would learn. Note that I have not included the full descriptors of the objects in order to save space.

Initial Percept

```
isa[table], supports[cup#tag0#, saucer]
isa[floor], supports[book, table], colour[white]
isa[cup], on[table], clean[yes], colour[white], in[kitchen]
isa[saucer], on[table], in[kitchen]
```

```
isa[book], on[floor], clean[no], colour[brown]
isa[robot], in[kitchen]
```

Action `PickUp cup`

Resulting Percept

```
isa[table], supports[saucer]
isa[floor], supports[book, table], colour[white]
isa[cup], in[robot], clean[yes], colour[white]
isa[book], on[floor], clean[no], colour[brown]
isa[robot], in[kitchen], contains[cup#tag0#]
```

Rule Learned

```
PickUp
Precondition
ARG0:  isa[cup], on[table], in[kitchen], colour[white],
       clean[yes],
OBJ1:  isa[robot], in[kitchen]
OBJ2:  isa[table], supports(Anything)#tag0#[cup, saucer],

Postcondition
ARG0:  -in[kitchen] -on +in[robot]
OBJ1:  +contains[#tag0#]
OBJ2:  -supports[#tag0#]
```

The object OBJ1 is the one that plays the role of receiving the object being picked up, the object OBJ2 is the one that plays the role of initially supporting the object being picked up. No other objects are included in the rule because no other object has changed.

Percept

```
isa[table], supports[saucer], colour[green]
isa[floor], supports[book#tag0#, table], colour[white]
isa[cup], in[robot], clean[yes]
isa[saucer], on[table], in[kitchen]
isa[book], on[floor], in[kitchen], clean[no], colour[brown]
isa[robot], in[kitchen], contains[cup]
```

Action `PickUp book`

Resulting Percept

```
isa[table], supports[saucer], colour[green]
isa[floor], supports[table], colour[white]
isa[cup], in[robot], clean[yes]
isa[saucer], on[table], in[kitchen]
isa[book], in[robot], clean[no], colour[brown]
isa[robot], in[kitchen], contains[cup, book#tag0#]
```

Rule Generalized

```
PickUp
Precondition
```

```
ARG0:  isa(Anything)[cup, book], on(Anything)[table, floor],
       in[kitchen], clean(Anything)[yes, no],
       colour(Anything)[white, brown]
OBJ1:  isa[robot], in[kitchen]
OBJ2:  isa(Anything)[table, floor],
       supports(Anything)#tag0#[cup, saucer, book],
       colour(Anything)[brown, white]

Postcondition
ARG0:  -in[kitchen] -on +in[robot]
OBJ1:  +contains[#tag0#]
OBJ2:  -supports[#tag0#]
```

Some of the objects in the preconditions have been generalized to cover both positive examples. Note that the robot object has not changed because only the "isa" and "in" attributes were common to both the rule and the matched object. The postcondition has not changed because the calculated effect is exactly the same as the one stored in the rule.

Percept

```
isa[table], colour[green]
isa[floor], supports[table#tag0#], colour[white]
isa[cup], in[robot], clean[yes]
isa[saucer], on[table#tag0#], in[kitchen]
isa[book], in[robot], clean[no], colour[brown]
isa[robot], in[kitchen], contains[cup, book]
```

Action `PickUp table`

Resulting Percept

```
isa[table], colour[green]
isa[floor], supports[table#tag0#], colour[white]
isa[cup], in[robot], clean[yes]
isa[saucer], on[table#tag0#], in[kitchen]
isa[book], in[robot], clean[no], colour[brown]
isa[robot], in[kitchen], contains[cup, book]
```

Rule Generalized

```
PickUp
Precondition
ARG0:  isa(Anything)[cup, book], on(Anything)[table, floor],
    in[kitchen], clean(Anything)[yes, no],
    colour(Anything)[white, brown]
OBJ1:  isa[robot], in[kitchen]
OBJ2:  isa(Anything)[table, floor],
    supports(Anything)[cup, saucer, book#tag0#],
    colour(Anything)[brown, white]

Postcondition
```

```
ARG0:  -in[kitchen] -on +in[robot]
OBJ1:  +contains[#tag0#]
OBJ2:  -supports[#tag0#]
Unless
ARG0:  isa[table], supports[saucer]
OBJ1:  []
OBJ2:  []
```

The action did not change the world in any way. EARL-1 records this as a failure. The reason that the action failed was because EARL-1 attempted to pick up an object (the table) that was supporting another object. The unless clause records everything that was true about the relevant objects when the action failed, but was not true when the action succeeded. There are only two possible reasons for the failure (assuming that the attributes are independent of each other): the object being picked up was a table, or because it supported a saucer.

4.7 Limitations

There are several significant limitations to EARL-1 that lead to the design of EARL-2 and EARL-3 described in the following chapters.

4.7.1 The agent cannot deal with conditional effects

EARL-1 assumes that an action can have only one kind of effect or no effect at all (when the action fails). In the real world, most actions can have a lot of different effects depending on their preconditions. Crane World does have several actions that have conditional effects. However, most of these conditional effects deal with negative effects (e.g. when picking up an object A from object B then if B has a contains attribute, then the "contains" attribute will be deleted, but if B has a supports attribute, then the supports attribute will be deleted). EARL-1 can handle such conditional effects by simply aggregating them into one postcondition. If the object has the attribute then it is removed, if the object does not have the attribute then there is no effect. However, there are a few positive effects in Crane World that cannot be simply aggregated in this way. An example of this is the PutDown action/rule. If EARL-1 puts down an object on the floor then one of the effects is +in[kitchen] (assuming that the robot is in the kitchen). But if EARL-1 puts the object down into a container c, then the effect would be +in[c] (e.g. +in[sink], or +in[box]). When generalizing postcondtions EARL-1 just aggregates these changes into one postcondition. But this can result in a set of effects such as: +in[kitchen], +in[sink], +in[box]. If such an effect were applied to an object then it would end up being in three places at once! The only reasonable way of solving this problem is for the agent to learn conditional effects.

4.7.2 The agent cannot guide its learning with rules it has already mastered

EARL-1 does not know how to plan using previously learned rules. For example, suppose that it knew that it needed to be holding a key to unlock a box and it had already learned a very reliable rule for picking up. If it were currently not holding a key but there was one lying on the table, it would not be able to plan (using the pick up rule) to pick up the key and use it to complete the unlock rule. EARL-1 would remain focused on trying to learn the unlock rule and would never consider any other actions. It would see that it is not holding the key and correctly predict failure. It would do this five times, become confident about the rule, and switch its focus of attention to a new action. EARL-1 needs to use its current knowledge to help it explore new actions.

The above scenario illustrates that the way in which EARL-1 gains confidence needs to be changed as well, EARL-1 should only be confident about a rule if it is confident both about the main clause and the unless clause.

4.7.3 The agent cannot deal with hidden effects

Some actions have hidden effects that EARL-1 cannot perceive directly. Locking and unlocking a door produce no visible effect that EARL-1 can perceive. If it were to execute the action, it would perceive no change and it would assume that the action failed. Currently it would continue unlocking the door, see no change, and "correctly" predict failure, until it became confident about the rule. This is obviously undesirable. A mechanism for dealing with actions that have hidden effects needs to be added. Also the naive assumption that an action failed if it had no visible effects is clearly not suitable for more complicated actions.

Chapter 5

Learning Agent 2: Dealing with Conditional Effects

5.1 Overview

One action can have many different effects depending on the precondition. For example:

If the agent drops a rubber ball then the ball bounces.
If the agent drops an egg then the egg breaks.
If the agent drops a pillow then the pillow will just be on the floor.

The agent should be able to learn a rule for the Drop action corresponding to each of these effects. For example, if the object being dropped is made out of rubber, then it will bounce, if it is fragile then it will break, otherwise it will end up on the floor.

A single action can also cause multiple effects, each related to different subsets of the percept. For example, suppose that there is a cup filled with water sitting on a table. Next to the cup is a fragile egg. If the robot picked up the cup (and in the process spilled some water and knocked the egg off the table), the effects would be that the cup would no longer be on the table, it would be in the robot's hand, there would be water on the floor, and the egg would be broken.

This action really caused three separate effects: the cup moving from the table to the robot's hand, the egg breaking, and water being spilled on the floor. The reason that the egg broke is because it was fragile and next to the object being picked up, everything else in the percept is irrelevant to this effect. The reason that there is water on the floor is because the cup contained water, everything else in the percept is irrelevant. The agent should be able to learn a rule for the Pick Up action corresponding to each of these effects. For example, if the object being picked up contains water, then there will be water on the floor, if there is a fragile object next to it then it will break, and the object being picked up will end up in the robot's hand.

I have extensively changed the rules and the rule learning algorithm that EARL-1 uses to allow the agent to learn about conditional effects. This second version of the agent (EARL-2) associates a list of sub-rules with each action type. Each sub-rule corresponds to a conditional effect. A sub-rule consists of a precondition and a postcondition. The (conditional) postcondition holds if its corresponding precondition holds.

5.2 Rules Learned

5.2.1 Each rule now consists of sub-rules

For each action type there is still one rule associated with it. However, each rule is made out of one or more sub-rules. Each sub-rule consists of a precondition and a postcondition. For each sub-rule, if its precondition holds, then so does its postcondition. This means that for a given percept, more than one sub-rule can hold. EARL-1's rules consisted of a mapping from a list of objects in the precondition to a list of objects in the postcondition. EARL-2 constructs its rules differently. In each sub-rule, there is no longer a direct mapping between objects in the preconditions and changes in the postcondition. Instead, each object in the precondition and in the postcondition is indexed by its type (the role it plays in the action). An object's type is either ARG# (if the object is an argument (# stands for the argument number)), or the role it played in the action (a list of the tags and what attribute each tag was in before and after the action was executed). For example the object type:

```
Contains[#tag0#]|Supports[#tag1#]
```

indicates that the object with this type contained the first argument of an action before the action was executed, and then supported the second argument of the action after the action was executed but no longer contained the first argument.

5.2.2 The precondition of each sub-rule is a disjunction of conjuncts

A precondition of each sub-rule is now just one long disjunction. A precondition matches a percept if at least one of its disjuncts matches the percept. Each disjunct is a list of conjuncts (a disjunct holds/is true if all of its conjuncts are true). A conjunct can be either an attribute with a value indexed by an object type, or a negation of a sub-rule. A conjunct consisting of an attribute with a value (indexed by an object type) matches a percept if the percept contains an object with the given type that has the same attribute with the same value.

In the preconditions, a conjunct can also directly negate a sub-rule, for example NOT RULE1. This conjunct holds/is true if sub-rule 1 is not true. So when checking the validity of this conjunct, EARL-2 has to first check the validity of sub-rule 1. This in turn can lead to more conjuncts that negate other sub-rules that also have to be checked, leading to a chain of sub-rules that have to be evaluated. If there is a loop in the chain of sub-rules (eg. a conjunct in rule1 negates rule2 and a conjunct in rule2 negates rule1) then the originating conjunct fails. When EARL-2 evaluates a precondition, it always checks disjuncts that do not contain any negations first.

An example of a precondition in a sub-rule could be:

```
Rule #1: PickUp
SUBRULE1:
   PRE:
   [(ARG0)(Clean, [Yes])] OR
   [([(Supports, [#tag0#])]|[])(Colour, [blue, white, brown])] OR
   [(ARG0)(isa, [Cup]),
      (Supports, [#tag0#])]|[])(isa, [saucer])] OR
   NOT RULE2
   POST: ...

SUBRULE2:
   PRE:
```

```
[(ARG0)(isa, [Door, Floor])] OR
[([(Supports, [#tag0#])]|[])(Clean, [No])]
POST: ...
```

This precondition would hold if the object being picked up was clean, OR the object supporting the object being picked up was blue or white or brown, OR the object being picked up was a cup AND the object supporting the cup was a saucer, OR if sub-rule 2 did not hold (i.e. the object being picked up was not a door and not a floor and the object supporting the object being picked up was not dirty).

5.2.3 There is a"dropped" clause associated with every precondition

As EARL-2 refines its rules, it drops disjuncts from the preconditions in the various sub-rules. The "dropped" clause associated with each precondition stores all the information that was dropped from the precondition. EARL-2 uses this "dropped" clause to help it match objects and to prevent disjuncts that had already been dropped from being put back into the precondition (see 5.4.3). Note that the "dropped" clause does not just blindly store what was dropped (that would be impractical as the list of dropped attributes would be enormous). Instead, it generalizes the dropped attributes similar to the way that EARL-1 generalized its preconditions.

5.2.4 The postconditions are disjoint from each other

EARL-2's postconditions for each sub-rule are analagous to the postconditions of each rule that EARL-1 learned. However, when EARL-2 refines its sub-rules, it makes sure that the postconditions are all kept disjoint from one another. Doing this greatly reduces the number of sub-rules it has to learn, as well as improving their flexibility. For example if there were three kinds of effects E1, E2, and E3 for a given action, then EARL-2 only needs to learn three sub-rules corresponding to each effect. If the postconditions were not kept separate, then EARL-2 would have to learn nine separate sub-rules, each corresponding to the different ways of combining the three effects

5.3 Learning a New Rule

EARL-2 learns a new rule in a very similar way to EARL-1. It matches objects in the before percept to those in the after percept, and uses the same two simple heuristics to eliminate irrelevant objects. However, it constructs the precondition and effects of the action differently. After the objects in the before percept and the after percept have been matched, EARL-2 computes the list of changes that each object underwent. Each list of changes (specific to each object that changed) is indexed by the object's type (see 5.2.1) and placed in the postcondition. EARL-2 determines the precondition by simply collating all of the attribute values pairs (corresponding to the changed objects) from the initial percept and making them a list of disjuncts. In this way, EARL-2 generates a new sub-rule for the action.

5.4 Refining Stored Rules

Once EARL-2 has generated a new sub-rule for this action type, it checks to see whether it already has any stored sub-rules. If it does, then EARL-2 generalizes those sub-rules that have the same effect as the new sub-rule and specializes those that have different effects.

5.4.1 The agent matches the objects in the new rule to the objects in the corresponding stored rule

EARL-2 does this in a similar way that EARL-1 did (see 4.4.1). However, when a best-match is required, EARL-2 behaves slightly differently. Because EARL-2 drops disjuncts from the preconditions in the sub-rules when refining rules (see 5.4.3), crucial information needed for matching may be lost in the process. That is why, when performing a best-match, EARL-2 uses information that is both in the precondition and in the "discarded" clause. EARL-2 retrieves all of the attribute values pairs that are in the precondition and the discarded clause and collates them into a single "object". It can then use this "object" to perform a best match.

5.4.2 The agent makes sure all of the postconditions are disjoint

To refine the stored rules, EARL-2 first scans through the entire list of stored sub-rules and checks whether or not any of the stored postconditions are actually compound ones that should be split into separate sub-rules. EARL-2 scans through the list of sub-rules and notes the differences between their postconditions and the new postcondition it calculated in the previous step. If a stored postcondition and the new postcondition are disjoint, equal, or the stored postcondition is a subset of the new postcondition, then the stored postcondition remains unmodified (because there is no reason to split the stored postcondition).

If the stored postcondition and new postcondition overlap or if the stored postcondition is a superset of the new postcondition, then this means that part of the stored postcondition is separable and belongs in a separate (new) sub-rule. EARL-2 thus splits the sub-rule with the given postcondition into two new sub-rules each having a part of the original stored postcondition. One of the new sub-rules gets everything in the original stored postcondition that was not in the new postcondition. The other sub-rule gets what was in both the original stored postcondition and in the new postcondition. Both of the new sub-rules keep the same precondition because the agent cannot determine which parts of the precondition caused which parts of the postcondition, so it plays it safe and associates the entire precondition with both of the new sub-rules.

Every time that the new postcondition is compared with a stored postcondition, EARL-2 removes the intersection of the two from the new postcondition. After EARL-2 finishes scanning through the list of sub-rules, anything that is left in the new postcondition is a completely new effect that belongs in a separate sub-rule. So EARL-2 adds a new sub-rule having the newly calculated precondition and whatever remained of the new postcondition as its postcondition. EARL-2 then removes the intersection of every precondition (in the stored sub-rules) and the new precondition from the new precondition, because the calculated postcondition is completely new and none of the stored disjuncts could possibly have caused it.

In this way all of the postconditions in the sub-rules are kept disjoint from one another.

5.4.3 The agent keeps the preconditions consistent with the new effect

After keeping the postconditions disjoint and adding any new effects, EARL-2 scans through the list of sub-rules again, to make sure that the precondition of each sub-rule is consistent with the example it calculated in the first step. It compares each sub-rule's postcondition with the original newly calculated postcondition. If a stored postcondition is a subset of the new postcondition, then this means that this sub-rule's precondition was true for this example and it should be generalized with the newly calculated precondition.

EARL-2 generalizes the stored precondition by adding all of the disjuncts in the newly calculated precondition to the stored precondition. Before adding a disjunct, EARL-2 checks the “dropped” clause to make sure that there are no disjuncts there that would subsume the disjunct being added. If there is such a discarded disjunct, then EARL-2 does not add it to the stored precondition as it has already established that it is irrelevant.

If there are “matching” disjuncts in the stored precondition and the new preconditions then they are “merged” into one disjunct that is placed into the stored precondition. Two disjuncts match if they have the same number of conjuncts and all of their conjuncts match. Two conjuncts match if their attribute values pairs have the same attribute name or they negate the same rule. EARL-2 merges two disjuncts by merging all of their conjuncts. It merges two conjuncts by combining all of the values (from two attribute values pairs with the same name) into a new conjunct. This merging is a form of generalization. Rather than having a long list of examples (e.g. Contains[A], Contains[B], Contains[C], etc.) EARL-2 creates a generalized version (e.g. Contains[Anything]). For example, EARL-2 would merge:

```
[(ARG0):(Clean, [No])] OR
[(ARG0):(Colour, [blue, white])] OR
[(Supports, [#tag0#])]|[]):(Clean, [yes]), NOT RULE2] OR
[([(Supports, [#tag0#])]|[]):(Colour, [brown]),
   (ARG0):(isa, [dish])]
```

and

```
[([(Supports, [#tag0#])]|[]):(On, [Floor])] OR
[(ARG0):(Colour, [brown])] OR
[(Supports, [#tag0#])]|[]):(Clean, [no]), NOT RULE2]
```

into

```
[(ARG0):(Clean, [No])] OR
[(ARG0):(Colour, [blue, white, brown])] OR
[([(Supports, [#tag0#])]|[]):(On, [Floor])] OR
[(Supports, [#tag0#])]|[]):(Clean, [yes, no]), NOT RULE2] OR
[([(Supports, [#tag0#])]|[]):(Colour, [brown]),
   (ARG0)(isa, [dish])]
```

If a stored postcondition is disjoint from the new postcondition, then this means that any disjuncts that are in both the corresponding stored precondition and the new precondition should be removed from the stored precondition. This is because the disjunct was true in the new precondition but the effect of the stored postcondition did not occur (since the two postconditions are disjoint), therefore the disjunct should not be in the precondition of the stored sub-rule.

5.4.4 If a precondition collapses, the agent considers more complex disjuncts

The above algorithm will work only if all of the conjuncts are independent of each other. If a conjunction of size greater than one is needed, then the algorithm will fail. This is because of the way disjuncts are eliminated when the rule is specialized.

For example, to start the car there has to be petrol in the gas tank and one has to have the right key. If a person started the car with the right key and petrol and then tried to start the car with the right key but no petrol (and it did not start), then the person would conclude (if they were using EARL-2’s algorithm) that having the right key is not the reason that the car started (since this disjunct was present both when the car started and when it did not). If the person then tried to start the car with petrol but without the right key, they would conclude that petrol was also not the reason the car started. They would conclude that neither having the right key nor there being petrol in the car was the reason the car starts.

To solve this problem, I modified the algorithm so that EARL-2 initially assumes that all the disjuncts are independent. Then, if the precondition collapses (i.e. all of the disjuncts have been eliminated), EARL-2 considers more complex rules by increasing the "depth" of each disjunct to consider disjuncts with two conjuncts in it. EARL-2 also considers conjuncts that are the negation of other sub-rules by introducing the negation of every other sub-rule as a possible conjunct (NOT RULE1, NOT RULE2, etc.).

If this two-deep precondition collapses again, then EARL-2 increases the "depth" again, to consider disjuncts with three conjuncts in it. Whenever a precondition collapses, EARL-2 increases its depth.

5.5 Exploration Strategy

EARL-2 uses a very similar exploration strategy of EARL-1 (see 4.5), but making and checking predictions is more complex.

5.5.1 The agent checks its prediction against the current percept

EARL-2's prediction is correct if and only if it matches the new percept exactly. Because EARL-2 can handle conditional effects, it can deal properly with multiple object deletion and addition. If different objects get added for a particular action, then EARL-2 does not have to generalize them. Instead, it can treat these effects as conditional effects and separate them into separate sub-rules. As a result, EARL-2 does not need the"add many" and "delete many" flags that EARL-1 needed, and it can (in theory) make perfect predictions.

5.5.2 The agent makes a prediction

Like EARL-1, the way that EARL-2 checks the accuracy of its rules, is by using them to make predictions of the outcome of its actions, and then comparing the predictions to the actual results. Given a starting percept, an action, and a set of rules, EARL-2 can make a prediction of what the resulting percept will be. It does this by first retrieving the rule corresponding to the action. It then scans through each of the sub-rules and checks whether the current percept matches their precondition (see 5.4.1). For every precondition that matches the current percept, EARL-2 uses the corresponding postcondition to modify a copy of the current percept. After being modified by all of the applicable sub-rules, this percept becomes EARL-2's prediction. EARL-2 then executes its action, receives the new percept, and starts the rule learning cycle again.

5.6 Example

This is an example of how EARL-2 generalizes its rules. I use letters to stand for attributes in order to make the algorithm clearer. $A \supset B$ means that A is the precondition and B is the postcondition of a sub-rule. I represent the before states and the effect (the difference between the before and after state) as attributes (letters) enclosed in parentheses.

Initial Rules: $A \vee B \vee C \vee D \supset P \wedge Q$
$E \supset R$

Before State: (C, D, E)
Effect: (R)

From this we can conclude that C and D definitely don't cause $P \wedge Q$, otherwise (C,D,E) would have had an effect of (R,P,Q). So we can remove C and D from the first rule. We also have to generalize the sub-rule for R. Note that we have to include C and D as possible reasons for R despite the fact that they were in the rule $A \vee B \vee C \vee D \supset P \wedge Q$ whose effect does not include R. This is because $A \vee B \vee C \vee D \supset P \wedge Q$ may have in the past been split from some rule that had R as part of its effect.

New Rules: $A \vee B \supset P \wedge Q$
$C \vee D \vee E \supset R$

Before State: (A, F)
Effect: (P)

The effect $(P \wedge Q)$ should be split into two sub-effects and F should be added as a possible cause of P. We can also conclude that A cannot be the cause of Q, otherwise the effect of (A, F) would have been (P, Q).

New Rules: $A \vee B \vee F \supset P$
$B \supset Q$
$C \vee D \vee E \supset R$

The above algorithm will work only if all of the variables are independent of each other (ie. no conjunctive rules of the form $((A \wedge B) \supset P)$ are required). If a conjunction is needed, then the algorithm will fail.

Suppose that $A \supset X$ and $B \supset X$ and $(A \wedge B) \supset E$ are the true rules that should be learned.

Before State: (A, B)
Effect: (E)

New Rule: $A \vee B \supset E$

Before State: (A)
Effect: (X)

New Rules: $B \supset E$ (A can't be responsible for E)
$A \supset X$

Before State: (B)
Effect: (X)

New Rules: $\phi \supset E$ (B can't be responsible for E)
$A \vee B \supset X$

Now the preconditions of effect E have become null, and the algorithm collapses. To solve this problem, whenever a rule's preconditions become null, EARL-2 increases the rule's complexity by one level. EARL-2 will now consider disjuncts that are one conjunct longer.

Before State: (A, B, C)
Effect: E
New Rule: $A \vee B \vee C \vee (A \wedge B) \vee (A \wedge C) \vee (B \wedge C) \supset E$

The following is an example of an actual rule that EARL-2 learned after picking up some sugar, a pot, attempting to pick up a table, and a fragile cup.

```
Rule #1:     PickUp
SUBRULE0:
        PRE:
        [(ARG0):(Contains, [Water])] OR
        [(ARG0):(Fragile, [Yes])] OR
        [(ARG0):(On, [Table])] OR
        [(ARG0):(Open, [Yes])] OR
        [([(Supports, [#tag0#])]|[]):(Clean, [yes])] OR
        [([(Supports, [#tag0#])]|[]):(Colour, [brown])] OR
        [([(Supports, [#tag0#])]|[]):(On, [Floor])] OR
        [([(Supports, [#tag0#])]|[]):(isa, [Table])] OR
        [([]|[(Contains, [#tag0#])]):(Contains, [Sugar])]
                DROPPED ([(ARG0):(Clean, [yes])],
                         [(ARG0):(In, [Kitchen])],
                         [([(Supports, [#tag0#])]|[]):(In, [Kitchen])],
                         ...
        POST:
        TYPE ARG0:    [In[+Robot], In[-Kitchen], -On]
        TYPE [(Contains, [#tag0#])]|[]:    [Contains[-#tag0#]]
        TYPE [(Supports, [#tag0#])]|[]:    [Supports[-#tag0#]]
        TYPE []|[(Contains, [#tag0#])]:    [Contains[+#tag0#]]
SUBRULE1:
        PRE:
        [(ARG0):(Supports, [Box, Coffee, Coffee, Cup])] OR
        [(ARG0):(isa, [Table])]
                DROPPED ([(ARG0):(Clean, [yes])],
                         [(ARG0):(Colour, [brown])],
                         ...
        POST:   No Changes
SUBRULE2:
        PRE:
        [(ARG0):(Clean, [No])] OR
        [(ARG0):(Colour, [white])] OR
        [(ARG0):(Fragile, [Yes])] OR
                DROPPED ([(ARG0):(In, [Kitchen])],
                         [(ARG0):(On, [Table])],
                         [(ARG0):(Open, [Yes])],
 ...
        POST:
        TYPE ARG0:    [Broken[+Yes], Fragile[-Yes]]
```

The sub-rule that captures the concept of failure (sub-rule 1) is almost perfect. It expresses it as a conditional rule: if the object being picked up is a table or it supports something, then there will be no result (failure). Sub-rule 2 captures the conditional effect of breaking an object: if the object being picked up is not clean or it is white or it is fragile then it will break. The precondition, though quite good, has two incorrect disjuncts. The reason for this is that the dirty white cup EARL-2 picked up was the only fragile thing it ever picked up. So it is reasonable for it to believe that dirty or white things will break when picked up. If it picks up a dirty, white object in the future that does not break, then it will remove these two incorrect disjuncts to form a perfect rule. Sub-rule 0 captures the default case of picking up. Its disjuncts get eliminated only very slowly. This is because the default case fails only rarely (when EARL-2 attempts to pick something up that is supporting something else, for example). The true precondition of the first sub-rule should just be NOT RULE 1. Despite the fact that sub-rule 0 is quite incorrect, as a tool for making predictions it is still quite useful. It is disjoint from sub-rule 1 and it is so general that it will tend to be true whenever sub-rule 1 isn't, and that is exactly how it should behave.

Note the EARL-2 never makes the same mistake twice. If it tried picking up a saucer supporting a cup that was on a table, then [(ARG0)(On, [Table])] would be met and sub-rule 0 would hold, [(ARG0)(Supports, [Box, Coffee, Coffee, Cup])] would also be met and sub-rule 1 would also hold. EARL-2 would predict that the saucer would be picked up and that there would be no changes. In conflicting situation like this, EARL-2 simply adds the changes together, so it would incorrectly predict that the saucer would be picked up. The actual result of the action would be no changes, so EARL-2 would remove [(ARG0)(On, [Table])] from sub-rule 0, and it would not make that mistake again.

5.7 Limitations

5.7.1 The agent does not generalize deletions

A minor problem is that EARL-2 forms a completely new sub-rule if there is a new effect. This is almost always the correct thing to do. However, sometimes it should generalize an existing effect rather than creating a new sub-rule. For example, if EARL-2 has a stored conditional effect of deleting a cup object and it sees a new effect that is the deletion of a plate, it would create a separate conditional effect for each deletion. Ideally it should generalize its conditional effect to be the deletion of a dish. However, it is unclear how similar two effects should be for the agent to generalize one conditional effect rather than creating a new sub-rule.

5.7.2 The agent cannot handle hidden effects

Like EARL-1, EARL-2 still cannot deal with hidden effects (see 4.7.3).

5.7.3 The exploration strategy is flawed

EARL-2 uses EARL-1's exploration strategy, but this is not really appropriate to rules that have sub-rules in them. In order to be confident about a rule, the agent should really be confident about every single sub-rule within the rule. However, EARL-2 becomes confident about a rule simply when its predictions for that rule succeed five times in a row. This means that if the action has three sub-rules, EARL-2 could become confident about the entire rule if it had only one well-learned sub-rule. For example, if EARL-2 had a good sub-rule for failure (trying to pick up something that has something else on top of it), it would become confident about the pick up rule by simply trying (and failing) to pick up five things that had something on top of them, even if it knew almost nothing about picking up fragile objects and the default pick up case.

5.7.4 It takes a long time to learn complex disjuncts

In order to learn a disjunct that requires three conjuncts, EARL-2 must first eliminate all of the singleton disjuncts, then it must eliminate every single disjunct with two conjuncts. In general, to learn a disjunct with n conjuncts, EARL-2 must first eliminate every set of disjuncts having fewer conjuncts, this is exponential in the length of n. This means that it is impractical for EARL-2 to learn preconditions that have disjunctions with long conjunctions in them.

Chapter 6

Learning Agent 3: Dealing with Hidden Effects

6.1 Overview

The third version of the agent that I have implemented (EARL-3) can now, with the assistance of a teacher, learn about hidden effects. EARL-3 learns almost the same rules that EARL-2 did and it uses almost the same algorithm. However, I have significantly altered EARL-2's exploration strategy so that EARL-3 can not only learn hidden effects, but also plan to a limited extent.

6.1.1 Hidden effects are extremely difficult to learn

Actions such as locking/unlocking a door have no visible effect that the agent can see. They still affect the world, but the agent can only see those effects by examining other actions. In this case, locking/unlocking a door affects whether or not the agent can open the door at a later time. The action is further complicated by the fact that it may have its own preconditions that must be satisfied as well. Note that it is extremely difficult to establish what the preconditions of an action with hidden effects are because it is difficult to determine what the effects of such an action are. In this example, the agent must be holding a key to successfully unlock a door. This means that the agent has to learn a sequence involving three actions (Pick up a key if not already holding one, Unlock the door, and Open the door).

6.1.2 The agent requires teacher assistance to learn hidden effects

The only way that the unaided agent could possibly learn sequences of actions such as (Pick Up the right key, Unlock the door, Open the door), would be for it use a brute force approach such as trial and error. However, the number of possible sequences of actions is exponential in the length of the sequence. Because learning hidden effects is so difficult it is unreasonable to expect the agent to learn even short sequences of actions autonomously. Instead, it depends on the assistance of a teacher.

Note that people have similar difficulties with learning long sequences of effects that may or may not have hidden effects. For example, it is unreasonable to give a child (who has never baked anything before) a set of raw ingredients and expect it to discover how to bake a cake. But if the child is shown how to bake a cake, it can quickly grasp the concept.

6.1.3 Using variables for hidden effects is not a good solution

One way of learning about hidden effects is to use variables to represent the hidden effects of an action. Once the agent has determined that an action has hidden effects (see 6.4.3), then it could just add a variable to the effects of the action. For example, the effect of locking a door could be "hiddenEffect1". Although this may seem like a good solution, with hiddenEffect1 standing for the door being in a locked state, the number of variables quickly becomes overwhelming. If the agent now realizes that Unlock has a hidden effect as well, it must introduce a new variable to stand for the new hidden effect (the agent cannot know that all Unlock does is remove hiddenEffect1): hiddenEffect2. Now when the agent does the Lock action again, what should the effect be? The presence of the two hidden effects in the precondition could have some effect on the Lock action, so the agent must introduce a third variable: hiddenEffect3. It is clear that the number of variables to stand for the hidden effects will just continue to increase indefinitely.

The reason that the above method failed was that the Unlock action was considered in isolation. However, a hidden effect has no meaning if it is not associated with something visible. In this example, the Locking and Unlocking actions must be associated with the Open action. A better way of learning rules about hidden effects is to learn them as parts of a sequence of actions.

6.1.4 Planning is necessary in order to properly learn conditional and hidden effects

EARL-1's and EARL-2's learning strategies have involved simply focusing their attention on a single action at a time in order to try to learn as much about that particular action as possible. However, this strategy is too simplistic for learning conditional effects and completely inappropriate for learning hidden effects.

An action may have several conditional effects and thus several sub-rules associated with it. The agent should only be confident about the whole rule if it is confident about all of the sub-rules. The agent should split its attention among all of the sub-rules equally, to prevent the situation where the agent is really confident about one sub-rule and knows almost nothing about all the others. This means that the agent must keep track of which sub-rules it is confident about and try to test the other sub-rules it is uncertain about. Since the difference between the preconditions in the sub-rules involves the state of the world, the agent may need to use some sort of planning in order to favour some sub-rules over others.

Planning is even more important when learning hidden effects because they inherently involve sequences of actions. An exploration strategy that focuses on one action at a time and makes the agent select the same action type over and over until it is confident about it is completely unsuited to learning sequences of different actions.

Note that another benefit of the planning approach is that the agent can continue to be focused on an action even when that action is not available in the list of possible actions. For example, suppose that the agent is focused on the Put Down action. If the agent is not holding anything then it cannot put anything down (the Put Down action is not a possible action). However, if the agent is already confident about the Pick Up rule then it can form a two step plan to first pick up an object and then put it down somewhere, and in the process learn something more about the Put Down rule.

6.2 Rules Learned

The rules that EARL-3 learns are almost the same as those that EARL-2 learned (see 5.2). However, EARL-3 now keeps track of the number of times that each sub-rule has been visited. EARL-3 also allows rules to have entire actions in their preconditions (indicated by prevAction).

6.3 Learning Rules

The rule learning algorithm that EARL-3 uses is also quite similar to the one used by EARL-2 (see 5.4). However, EARL-3 no longer refines rules for action types that have hidden effects. This is because the preconditions and effects of hidden effects really span multiple actions, so the rule refining algorithm that EARL-2 used is not appropriate for these kinds of actions. Nonetheless, I found that the rules generated for actions with hidden effects (when EARL-3 examined the teacher's demonstrated sequence, see 6.4.4) were always almost perfect and did not require further refinements. The reason for this is almost certainly because Crane World is too simple.

6.4 Exploration Strategy

6.4.1 When the agent becomes confused, it asks the teacher for assistance

EARL-3 asks for help in the same way as EARL-2 did. EARL-3 continues to experiment with the world until it becomes confused with an action, because it repeatedly cannot predict the outcome of the action. Once EARL-3 is confused by an action it puts it aside and experiments with other actions. When EARL-3 is confused about some actions and confident about all others, it asks for assistance with those actions it is confused about.

While the teacher is demonstrating actions, EARL-3 delays generalization and simply remembers the sequence of actions demonstrated by the teacher. Once the teacher completes the demonstration, EARL-3 analyzes the sequence of actions demonstrated by the teacher and modifies its rules accordingly.

6.4.2 The agent assumes that the teacher is helpful

When the teacher demonstrates a sequence of actions, EARL-3 makes several assumptions about the teacher's assistance:

- The teacher demonstrates the sequence of actions only once and each action demonstrated is relevant to the sequence of actions being demonstrated. The teacher does not try to confuse the agent by showing it unimportant actions.
- The teacher demonstrates the actions in the correct order. For example, the teacher would pick up a key before unlocking the door.
- Whenever the teacher demonstrates an action with a hidden effect, the teacher will demonstrate that effect with another action to show what (hidden) effect the action had. For example, if the teacher demonstrates the Unlock action, then after unlocking the door the teacher would demonstrate that the door can now be opened.

6.4.3 The agent uses a few simple heuristics to decide which actions in the demonstrated sequence have hidden effects

When the teacher finishes demonstrating a sequence of actions, EARL-3 uses the following heuristics to determine which actions had hidden effects:

- An action at the end of a sequence does not have any hidden effects, otherwise the teacher would have demonstrated these effects with another action (Note that if this action is demonstrated again within another, later sequence of actions, then EARL-3 could still learn hidden effects for that action later).
- An action demonstrated by the teacher with no visible effects definitely has hidden effects, because the teacher would not demonstrate an action that failed.
- If the effect of an action in the sequence is surprising (the effect is different from the predicted one and the precondition of the action did not change), then the previous action had hidden effects that influenced this action.
- All the other actions have no hidden effects.

6.4.4 The agent incorporates actions with hidden effects into the preconditions of the subsequent actions

As mentioned above, EARL-3 assumes that any action with hidden effects that is demonstrated by the teacher will be demonstrated in a sequence of actions with some other action definitely following the action with hidden effects. The action following the action with hidden effects is the one that is affected somehow by the hidden effects of the previous action.

EARL-3 modifies its rules by adding the action with hidden effects to the precondition of the rule of the action that immediately followed it. The rule that EARL-3 learns is that it should always execute the specified action in the precondition before executing the action corresponding to the rule.

Note that this means that EARL-3 may perform unnecessary actions. It will always execute the action in the precondition even though it may not be necessary. For example, if the Unlock action was in the preconditions of the Open action (i.e. in order to open something you have to first unlock it), EARL-3 would always perform the Unlock action before doing the Open action even if the door was already unlocked. However, this is not a serious problem, because EARL-3 has no way of knowing that a door stays unlocked when it has been unlocked once (and at least EARL-3 can successfully unlock locked doors).

6.4.5 The agent now uses planning rather than a simple focus on action in its exploration strategy

When learning, EARL-3 still focuses on a single type of action at a time, however, it now combines this with a simple three step planner (a planner that produces plans that require no more than three steps). In each Action state, EARL-3 checks the rule of the action it is currently focused on. It then finds the sub-rule that it is least confident about (has been executed the least number of times) and tries to form a short plan to execute the sub-rule (meet the requirements of the sub-rule's precondition).

EARL-3 first checks whether or not the current state matches the sub-rule's precondition. If it does, then EARL-3 scans through the list of possible actions until it finds an action with an action type matching the one that it is focused on and selects it. If there are no actions with the appropriate action type, then for each sub-rule's precondition, EARL-3 removes the intersection of the precondition and the percept representing the current state from the precondition. It does this because each of the preconditions in the sub-rules predicted that there would be at least one action with the given type in such a state. Since there was no such action, the preconditions must be (partially) wrong. It then becomes confused about this action type and switches its focus to a different action type.

If the current state does not match the sub-rule's precondition, then EARL-3 scans through the list of possible actions again. It considers only action types for which it has a rule that it is already confident about. Once it finds an action type that it is confident about, EARL-3 retrieves the corresponding rule and checks what the effect of the action would be. It then checks whether the effect of the action would meet the sub-rule's precondition. If it does then EARL-3 selects this action. If it does not then it checks the next possible action. If it does not find any appropriate action, then it becomes confused about this action type and switches its focus to a new action type.

If the sub-rule's precondition includes an action with a hidden effect, then EARL-3 tries to recursively form a plan to execute the action with the hidden effect. It goes through exactly the same procedure (see above) to select an action, the only difference is that it tries to meet the precondition of the action with the hidden effect (rather than the precondition of the action following it). Once EARL-3 manages to execute the action with the hidden effect, the next thing it does is to immediately execute the action it was focused on (i.e. the one that had the action with the hidden effect in its precondition). This means that although EARL-3 can produce plans that are 3 steps long, it effectively uses a 2-step planner since no planning is involved between the second to last and last steps.

Note that since rules can now span multiple actions, I extended the tagging in the vision system to maintain tags over two actions. Initially (as before) a tag representing the first argument of an action looks like #tag0#. During the next action this tag becomes @oldTag0@ and #tag0# is applied to the argument of the new action. In the next action the @oldTag0@ disappears and the object becomes like every other object.

A sub-rule can only ever have an action type in its precondition (e.g. Unlock) not a specific action (e.g. Unlock door). This means that there can be multiple possible actions that match the action type in a precondition (e.g. Unlock door, Unlock box, Unlock car, etc.). If the last action executed was Unlock box, then EARL-3 should open the box to complete the action sequence (Unlock, Open) rather than trying to open the car or the door. EARL-3 uses the new tagging system to help it to know which action to choose. The box will have a @oldTag0@ attached to it while the other objects will not. EARL-3 just chooses the object that has an @oldTag@ attached to it.

An example of planning and learning hidden effects would be:

Suppose the Open action has two sub-rules associated with it:

a) The door opens successfully
b) The door fails to open

Suppose that EARL-3 is confident about/understands sub-rule b), but is not confident about sub-rule a). It would then form a plan to meet the preconditions of that sub-rule. The action Unlock door is in the precondition of the sub-rule (from a previous demonstration by the teacher), so EARL-3 would create the following initial plan:

..., Unlock door, Open Door

It would then try to satisfy the preconditions of the Unlock rule. The preconditions of the Unlock rule include the constraint that the Robot must be holding a key (from a previous demonstration by the teacher). EARL-3 would then scan through all of the rules that it is confident about to see whether any of their postconditions satisfy the preconditions of the Unlock rule. If EARL-3 were confident about the Pick Up rule it would note that one of the effects of picking up a key would be that the Robot would then hold the key (satisfying the Unlock rule). So it would form the plan:

..., Pick Up key, Unlock door, Open door.

EARL-3 would then check the preconditions of the Pick Up rule. If they were already satisfied then it would execute the plan, if they were not satisfied then it would stop searching for a plan (it would not attempt to try to form any plans longer than 3 steps) and try a different rule.

6.5 Example

This is an example of the rules EARL-3 learns. It learned the Pick Up rule by picking up a pot, some sugar, and then trying to pick up a table. Note that there is a number next to each sub-rule indicating how many times it has been "executed". At this point the teacher (me) intervened. I did not wait for EARL-3 to explicitly ask for assistance because it would not do so until it had become confident about all of the other action types (which would take a long time and not be relevant to this example). I demonstrated how to open a door by picking up the key, unlocking the door, and then opening it. EARL-3 correctly inferred that the Unlock action has hidden effects (because the teacher demonstrated the action and it had no visible effects). So it put the Unlock action into the precondition of the Open action, and it put the effects of picking up the key into the precondition of the Unlock action. Note that it really should put the entire effect of picking up the key into the precondition of the Unlock action but it could not do so due to the limitations of the vision system in Crane World (see 6.6.1).

```
Rule #1:     Open
SUBRULE0: (1)
        PRE:
        [(ARG0):(Clean, [yes])] OR
        [(ARG0):(Colour, [Brown])] OR
        [(ARG0):(In, [Kitchen])] OR
        [(ARG0):(On, [Floor])] OR
        [(ARG0):(Open, [No])] OR
        [(ARG0):(sight_isa, [Door])]
        POST:
        TYPE ARG0:    [Open[+Yes], Open[-No]]
PrevAction: Unlock


Rule #2:     PickUp
SUBRULE0: (2)
        PRE:
        [(ARG0):(Colour, [Iron])] OR
        [(ARG0):(Contains, [Water])] OR
        [(ARG0):(On, [Table, Floor])] OR
        [(ARG0):(Open, [Yes])] OR
        [(ARG0):(sight_isa, [Key])] OR
        [([(Contains, [#tag0#])]|[]):(Contains, [Table@oldTag0@,
           Robot, Cup, Door, Key#tag0#, Box, Box, Sink, Coffee,
```

```
            Coffee, Floor])] OR
        [([(Supports, [#tag0#])]|[]):(Clean, [no, yes])] OR
        [([(Supports, [#tag0#])]|[]):(Colour, [Grey, brown])] OR
        [([(Supports, [#tag0#])]|[]):(Supports, [Table@oldTag0@,
            Robot, Door, Key#tag0#, Box, Sink])] OR
        [([]|[(Contains, [#tag0#])]):(Contains, [Sugar, CoffeePot])] OR
                DROPPED ([(ARG0):(Clean, [yes])],
                         [(ARG0):(In, [Kitchen])],
                         [([]|[(Contains, [#tag0#])]):(On, [Floor])],
 ...
        POST:
        TYPE ARG0:    [In[+Robot], In[-Kitchen], -On]
        TYPE [(Contains, [#tag0#])]|[]:    [Contains[-#tag0#]]
        TYPE [(Supports, [#tag0#])]|[]:    [Supports[-#tag0#]]
        TYPE []|[(Contains, [#tag0#])]:    [Contains[+#tag0#]]
SUBRULE1: (1)
        PRE:
        [(ARG0):(Supports, [Cup, Box, Coffee, Coffee])] OR
        [(ARG0):(sight_isa, [Table])]
                DROPPED ([(ARG0):(Clean, [yes])],
                         [(ARG0):(Colour, [brown])],
                         [(ARG0):(In, [Kitchen])],
                         ...
        POST:   No Changes


Rule #3:     Unlock
SUBRULE0: (1)
        PRE:
        [(ARG0):(In, [Robot])] OR
        [([]|[(Contains, [#tag0#])]):(Contains, [#tag0#])]
NextAction: Open
ARG0: Key
```

6.6 Limitations

6.6.1 The planning is not as sophisticated as it could be due to the limitations of the vision system in Crane World

When EARL-3 checks the effects of actions during the planning process, it should really apply the actions (mentally) to the current state and make a complete prediction as to what the new state would be. However, the vision system in Crane World only tags objects that take part in actions that are actually implemented. It does not allow any tagging of objects in actions that are merely "imagined". This means that when EARL-3 plans, it cannot attach tags to any objects and thus it cannot determine the roles that objects play in the action, which is critical in object matching when the agent makes a prediction (see 4.5.3). As a result, EARL-3 cannot make full predictions, instead it only considers the arguments of the action (ignoring other objects that may have been important). Fortunately this is not a big issue in Crane World, because the world is quite simple, and this kind of limited prediction is almost always sufficient.

Chapter 7

Conclusion

7.1 My Contributions

I have designed and implemented a system (EARL) that can learn STRIPS-like rules by experimenting with actions and matching their before and after states. It can use these rules to predict the effects of its actions and to make short plans, in order to facilitate its learning process.

7.1.1 I introduced a method of matching that recognizes role

When generalizing rules learned in a world that does not provide object identity (like Crane World), some form of object matching must be performed. The standard approach of doing a best match is inappropriate when learning rules about actions, because completely different objects should sometimes be matched on the basis of the role they played in the action.

I addressed this issue by modifying the vision system in Crane World to track the objects in the arguments of an action by attaching tags to them. I then used these tags to determine what relation the other objects had to the objects in the arguments. EARL was then able to match objects according to the role they played in the action rather than having to do a best match.

7.1.2 I developed an algorithm for learning conditional effects

Some actions can have multiple or different effects depending on their preconditions. Naively learning a single rule for each action type does not work for these more complicated actions. This is because each possible effect that the action can have has it is own precondition that should not be generalized with the preconditions of the other possible effects.

I tackled this problem initially by extending the STRIPS-like rules to have an unless clause to represent failure, a limited form of conditional effects. I extended this solution to handle arbitrary conditional effects by splitting single rules into a set of sub-rules where each sub-rule represents one conditional effect that the action can have. I represented each precondition of each sub-rule as a disjunction that EARL generalizes for each positive example and specializes for each negative example. To make sure that the rules were as flexible as possible I made sure that EARL kept the postconditions disjoint from each other.

7.1.3 I proposed a method of learning hidden effects

Hidden effects are extremely difficult to learn. This is because their effects cannot be observed immediately or directly. These effects must be deduced from the way they affect other actions. This involves learning a sequence of actions. Unfortunately the number of possible sequences of actions is exponential in the length of the sequence.

The method I used to solve this problem was for EARL to learn hidden effects with the help of a teacher. I make the reasonable assumption that the teacher is helpful and exploit this to make the exploration feasible. EARL learns independently until it becomes confused with an action. It then asks the teacher for assistance, observes the teacher demonstrate a sequence of actions once, and infers which actions have hidden effects, and then modifies its rules to incorporate the hidden effects into them.

7.1.4 I developed an exploration strategy that facilitates rule learning

The space of possible rules that the agent can learn is enormous. Searching it randomly means that it would take the agent a prohibitively long time before it could learn any useful rules (rules that can accurately predict the effects of actions). The agent explores the space of rules by performing actions that are themselves described by rules. Ideally the agent should use rules it is already confident about to guide its exploration.

I constrained EARL's exploration strategy by first making it focus on a single action type at a time, so that it would become confident about one rule before trying to learn others. I added a confidence and confusion factor to each rule so that the agent would not waste time learning rules it was already confident or confused about. To make sure that the agent examined each sub-rule equally, I added a planning mechanism to the agent so that it could use rules it was already confident about to form short plans to visit the sub-rule it was least confident about.

7.2 Limitations and Future Work

Despite the contributions of this project it has a number of limitations which give opportunity for future research.

7.2.1 Reactive rules are also needed in a dynamic world

The simulated world in which my agents learn is completely static. The world can only ever change as a direct result of an agent's actions. STRIPS-like rules with a precondition and effect are very appropriate to such worlds. However, Crane World is not very realistic in this regard. The real world is very dynamic and many changes in it (e.g. the weather) have nothing to do with our actions. These changes often occur very suddenly and they may require instantaneous responses. For example, if you are standing in the middle of a train track and a train is rushing towards you, you cannot just stand there for five minutes working out a plan for the optimal way of getting off the tracks.

Reactive rules are of the form: in situation A the agent should do action B. The effect is not specified at all. To decide what to do the agent just has to find which rule best matches the current situation and simply do the corresponding action. Note that no planning is required and the agent can react almost instantaneously. The down side of this approach is that the number of possible situations the agent can find itself is unlimited, and so the agent would require a very large number of reactive rules to handle all of them properly. If it had a small number of reactive rules then sometimes they would suggest useless or even harmful actions.

7.2.2 STRIPS-like rules and reactive rules are complementary

I believe that STRIPS-like rules and reactive rules should be combined within a single agent. The reactive rules can be used as heuristics to guide planning done with STRIPS-like rules [7], conversely STRIPS-like rules can also be used to guide reactive rules by doing a short look-ahead. STRIPS-like rules also provide a way of learning reactive rules that do not require any teacher. The agent could first learn STRIPS-like rules by experimenting with the environment, and then use those STRIPS-like rules to generate plans. It could then learn reactive rules by examining the plans that it formed.

7.2.3 It takes too long for the agent to learn a correct precondition

The preconditions that EARL-1 learned are not suitable for learning conditional effects. The disjunctive preconditions used by EARL-2 and EARL-3 are much better suited to learning conditional effects, however it takes EARL-2 and EARL-3 a long time before they can learn a complex conjunction (exponential in the length of the conjunction). Although this is not a problem in Crane World where the length of the conjunctive is never longer than two, this would sometimes be a limitation in the real world.

7.2.4 The rule learning needs to be extended to cover multiple actions

When EARL-3 identifies a hidden effect in an action (by examining a sequence of actions demonstrated by the teacher), it calculates the preconditions of that action (by looking at the previous action performed by the teacher) and the effects of the action (by storing the action in the precondition of the next action in the sequence) and then stops learning anything about that action in the future. This is not an issue in Crane World because the preconditions EARL-3 can calculate just by examining the actions of the teacher are almost perfect. However, in more complicated worlds, it would be more desirable for the agent to continue refining the preconditions and effects of hidden actions.

7.2.5 My agents cannot learn an adequate rule for the Move action

An issue that I did not address in this project is the learning of actions that have variable effects. Turning on a light in a completely dark room or moving to a new room are examples of this kind of action. Moving will have a different effect depending on the room to which the agent moves. Moving to the same room can also have a different effect. For example, if the agent removes some objects from the room, then the next time it returns the room will look different. Conditional effects cannot express these variable effects because they depend on the state the world was in an indefinite number of actions ago (in essence these are a kind of "hidden" precondition!). The effect of moving to the kitchen depends on the state the world was in the last time the agent was in the kitchen.

A possible solution would be for the agent to have a mental map of the world. As the agent explores the world, it could build up a mental image of the places and objects that it has seen. In this way it would be able to not only learn actions with variable effects such as the Move action, but be also able to learn actions that have remote effects (e.g. turning on a switch in one room causing a light to come on in another room). However, in a very rich world with a large number of objects it seems unreasonable for the agent to remember every single object that is in every single room. It is an open question of which objects it should remember and whether or not it should perform some generalization on the objects it does remember.

7.2.6 The vision system needs to be improved

The vision system in Crane World currently only tracks the objects in the arguments of an action that is actually taking place. It does not have the capability of tracking objects that are involved in hypothetical or predicted actions. I believe that this is an unreasonable limitation on the agent. The agent should be able to reliably predict the outcome of actions without actually having to do them (currently the agent makes a prediction as it executes an action). Note that in order to properly plan, an agent must be able to make hypothetical yet reliable predictions of what the effects of various actions would be.

7.2.7 Object types and classes should be added to the world

When generalizing its preconditions, EARL-1 assumed that if an attribute can have two possible values then it can have any value at all. Though this is a quite useful assumption to make in Crane World, it is certainly not true in the real world. A more flexible and powerful way of generalization would be to generalize values to the same class or type. For example, isa[cup] and isa[plate] could be generalized to isa[dish].

7.2.8 Crane World is still not complex enough

With the exception of the Move action, EARL-3 can learn very good rules for every action defined in Crane World. EARL-3 has the capability of exploring (almost) everything that there is to explore in Crane World. However, it is unclear how well this agent would do in a more complicated environment. In order to more fully test future agents, Crane World should be extended to have hundreds and even thousands of objects with many more actions that have conditional effects and hidden effects, as well as having complicated conjunctive and disjunctive preconditions.

Bibliography

[1] ANDREAE, J. H. *Associative Learning for a Robot Intelligence*. Imperial College Press, Imperial College, London, 1998.

[2] CHAPMAN, D. *Vision, Instruction, and Action*. The MIT Press, Cambridge, Massachusetts, USA, 1991.

[3] FIKES, R. E., AND NILSSON, N. J. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence 2*, 3/4 (1971), 189–208.

[4] RUSSELL, S., AND NORVIG, P. *Artificial Intelligence A Modern Approach*, Second ed. Pearson Education, Inc., Upper Saddle River, New Jersey, USA, 2003.

[5] RYSZARD S. MICHALSKI, JAIME G. CARBONELL, T. M. M. *Machine Learning An Artificial Intelligence Approach*. Tioga Publishing Company, Palo Alto, California, 1983.

[6] SEBASTIAN BADER, PASCAL HITZLER, S. H. The integration of connectionism and first-order knowledge representation and reasoning as a challenge for artificial intelligence. In *INFORMATION'2004* (Tokyo, Japan, 2004).

[7] SHANAHAN, M. Using reactive rules to guide a forward-chaining planner. In *6th European Conference on Planning* (2001).