Incremental Learning in Inductive Programming

Robert Henderson
University of Edinburgh, UK
robh93@googlemail.com

Abstract

Inductive programming systems characteristically exhibit an exponential explosion in search time as one increases the size of the programs to be generated. As a way of overcoming this, we introduce *incremental learning*, a process in which an inductive programming system automatically modifies its inductive bias towards some domain through solving a sequence of gradually more difficult problems in that domain.

We demonstrate a simple form of incremental learning in which a system incorporates solution programs into its background knowledge as it progresses through a sequence of problems. Using a search-based inductive functional programming system modelled on the MagicHaskeller system of Katayama (2007), we perform a set of experiments comparing the performance of inductive programming with and without incremental learning. Incremental learning is shown to produce a performance improvement of at least a factor of thirty on each of the four problem sequences tested. We describe how, given some assumptions, inductive programming with incremental learning can be shown to have a polynomial, rather than exponential, time complexity with respect to the size of the program to be generated. We discuss the difficulties involved in constructing suitable problem sequences for our incremental learning system, and consider what improvements can be made to overcome these difficulties.

Keywords Inductive programming, Inductive functional programming, Incremental learning

1. Introduction

Inductive Programming (IP) differs from more conventional machine learning techniques in that it features the use of a general, expressive programming language as a space of hypotheses for describing patterns in data. Herein lies both the attraction and the apparent downfall of IP: having such an expressive hypothesis space allows IP to be used to model complex or recursive patterns that simply cannot be represented with the more conventional methods (feedforward neural networks or decision trees, for example). On the other hand, this expressivity also means that IP methods can become intractable very quickly when applied to larger problems. State of the art IP systems such as ADATE (Olsson 1995), Igor II (Kitzelmann 2007), and MagicHaskeller (Katayama 2007) have shown promise on relatively simple arithmetic and list processing problems, but are not currently capable of synthesising the kinds of complex programs that realistic practical applications would demand. See Hofmann et al. (2009) for a recent evaluation of the capabilities of these systems.

How can we solve this dilemma, and get the benefits of a general, expressive hypothesis space as well as a method that is computationally tractable? It has been proposed (Solomonoff 2002; Schmidhuber 2004) that combining IP with *incremental learning* may provide a solution. An incremental learning system is one that can automatically modify its inductive bias towards a given domain through solving a sequence of successively more difficult problems in that domain. In other words, incremental learning is about gaining the expertise required to solve hard problems through the experience of solving easier ones. If successfully equipped with an incremental learning mechanism, a system should be able to learn to solve complex problems without the need for a human expert to hand-code extensive domain-specific knowledge or algorithms into its workings.

In this paper we present experimental evidence that incremental learning is a viable means for producing orders of magnitude performance improvements in IP. We start with a review of previous work in IP that features incremental learning (section 2). We then describe the particular incremental learning mechanism to be evaluated here (section 3), and give an overview of the IP system that was used in our experiments (section 4). We present the experiments themselves along with their results, and give an explanation for these results in the form of a computational complexity argument (section 5). Finally, we discuss the limitations of our chosen incremental learning mechanism, and consider what improvements are required before it can be of practical use (section 6).

2. Previous work

Quinlan and Cameron-Jones (1993) were probably the first to demonstrate a form of incremental learning in an IP context. They showed how their inductive logic programming system, FOIL, was able to solve more than half of the prob-

lems in a sequence of 18 textbook logic programming exercises presented to it in order of gradually increasing difficulty. This was made possible by having the system add each solution program to its background knowledge as it went along. It could therefore potentially re-use solutions to earlier problems as primitive elements in the construction of solutions to later problems.

More recently, Schmidhuber et al. (1997) studied an incremental learning mechanism which they termed 'adaptive Levin search'. The idea behind adaptive Levin search is that, in a search-based IP system, the inductive bias can be controlled by weighting the different programming language primitives according to how frequently they should be used. As a system solves a succession of problems, these weights are gradually modified according to how often each primitive actually occurs in solution programs. Thus, the system becomes biased towards re-using primitives that were present in successful programs in the past. Adaptive Levin search was shown to produce some performance improvement on a selection of simple problem sequences.

Schmidhuber (2004) later followed up the work on adaptive Levin search with a fully fledged incremental learning IP system called OOPS. OOPS supported both a weight modification mechanism with a similar role to the one in adaptive Levin search, as well as an ability to invoke chunks of code from past programs in solutions to new problems. However, in the problem sequence that Schmidhuber tested, which involved solving the general 'towers of Hanoi' problem, only the weight modification mechanism was shown to provide a direct performance benefit.

Khan et al. (1998) made a brief study into incremental learning in inductive logic programming, under the name of 'repeat learning'. Using the Progol inductive logic programming system, they demonstrated how helper predicates invented in order to solve one problem may be re-used when constructing the solution to another. They chose a problem domain concerning the inference of the general descriptions of moves in chess.

In this paper, we have chosen to focus on the kind of incremental learning mechanism that was employed in FOIL, that in which a system adds solution programs to its background knowledge as it progresses through a problem sequence. As we shall see, this simple method is remarkably powerful. The main drawback of Quinlan's and Cameron-Jones' short study is that they did not provide a direct comparison between scenarios with and without incremental learning. We shall remedy that with the experiments presented here.

3. Incremental learning mechanism

We aim to give a convincing demonstration of one simple but effective incremental learning mechanism. The mechanism works as follows: a sequence of successively more difficult, but related, problems is presented to an IP system. The system must solve the problems in the order given, and will incorporate each solution program into its object language as a new primitive function (i.e. into its background knowledge) as it goes along. This addition of these new functions to the system's object language is what constitutes the modification of its inductive bias. For an appropriately designed problem sequence, we would expect the time taken for the system to solve whole the sequence, with the help of incremental learning, to be much less than if it were tasked simply with solving the final problem of the sequence in isolation.

One can see how this mechanism might be expected to work effectively by considering how, particularly in functional programming, it is often natural to express the solution to a complex problem in terms of the solutions to one or more simpler problems already solved. This breaks the program down into smaller, more managable units, and is a technique commonly known as *procedural abstraction* when used by human programmers.

4. Implementation

We implemented, for the purpose of this study, a simple brute-force search based IP system modelled on the MagicHaskeller system of Katayama (2007). We shall refer to our implemented system as 'MagicLisper' (it was written in Common Lisp). In this section, we first review MagicHaskeller and explain our reasons for choosing it, then we describe how our system differs from MagicHaskeller in a few respects. We also talk through an example usage of our system on an IP problem.

4.1 Review of MagicHaskeller

MagicHaskeller is a search-based inductive functional programming system that infers programs from input-output training examples. Its main distiguishing feature is the brute-force algorithm that it uses to synthesise solution programs. More or less, it simply generates and tests all possible programs in its object language in order of length, using a breadth-first search, until it finds one that matches the training examples. This is tractable because of two features of MagicHaskeller's object language. Firstly, the language is strongly typed, with only type-consistent programs being considered by the search algorithm. Secondly, recursion is supported not explicitly, but via the use of certain of higher-order primitive operations known as *morphisms* (Augusteijn 1998). These morphisms are essentially generalisations of standard functional programming operations such as *map* and *reduce*, and with them, many useful recursive processes can be expressed concisely. Ultimately, these two features combine to produce a search space that contains rather few obviously useless programs, allowing brute-force search to fare well.

For this investigation into incremental learning, we chose to use a system based on MagicHaskeller for two reasons. Firstly, MagicHaskeller's search algorithm is fast;

synthesising simple recursive programs takes only a matter of seconds. Secondly, the search algorithm is simple and predictable; it is easy to understand exactly why MagicHaskeller succeeds or fails in finding a solution to a given problem, which helps immensely when one is designing problem specifications. It is for this second reason in particular that we chose MagicHaskeller as our base rather than an alternative such as ADATE or Igor II.

4.2 Differences between our system and MagicHaskeller

The object language of MagicLisper has the same form as the 'de Bruijn lambda calculus' language used in the version of MagicHaskeller described in (Katayama 2007). There is one significant structural difference: for the sake of simplicity, MagicLisper's type system does not support parametric polymorphism; instead, every primitive function in its object language has one or more explicit ground types. The default library of primitive functions and constants used by MagicLisper is given in table 1. Also see figure 1 for precise definitions of the morphism primitives.

In this paper we shall use a Lisp-style notation to represent programs. So, for example, the following program (`sum-elems`), which sums the elements of a list, in Haskell notation:

```
(\ a1 -> paralist (\ a2 a3 a4 -> + a4 a2) a1 0)
```

is written in the Lisp notation as:

```
(λ (a1) (paralist (λ (a2 a3 a4) (+ a4 a2)) a1 0))
```

MagicHaskeller searches through programs in order of length, or more precisely, it searches through programs in order of the total number of functor and lexical variable invocations they contain. In MagicLisper, we generalise on this process by requiring that primitive functors each be assigned a numerical weight. Programs are synthesised in order of total weight, this being the sum of the weights of their component functor and lexical variable invocations. Lexical variables always receive a weight of 0.4. The weights of the default primitive functors range between 2.0 and 4.5 (see table 1). As an example of how to calculate the total weight of a program, consider the `sum-elems` program mentioned above, which has a weight of 12:

	`paralist`	`+`	`a4`	`a2`	`a1`	`0`	Total
Weight	4.0	3.4	0.4	0.4	0.4	3.4	12

Note that symbols occuring in lambda parameter lists do not contribute to the calculation.

The weighting feature allows one to manually bias the system towards using certain primitives by assigning them lower weights. This extra flexibility allows our system to potentially handle a larger primitive library than MagicHaskeller, since more rarely used primitives can be given higher weights to minimise their negative impact on the search performance. Note that if one sets all the weights to the same value, our search algorithm reduces to that of MagicHaskeller. In this study, the weights were chosen by hand; however, we note that for a more advanced system it would make sense to have these weights tuned automatically. To justify our choice of weight values, we have tested MagicLisper's performance on a selection of nine non-incremental problems, both with and without the customised weights (table 2). The problems all exhibit a significant increase in solution speed due to the custom weights, ranging from a factor of 2.4 to a factor of 165.7.

MagicLisper does not employ the memoisation or fusion rule optimisations of MagicHaskeller. Finally, MagicLisper requires the user to explicitly specify the maximum number of 'steps' for which to test any candidate solution program on a given training example. Each step corresponds to one evaluation by the interpreter of a sub-expression within a program, and this 'number of steps' is an approximate specification of the maximum time to spend testing each program.

4.3 Example usage of our system

Let us briefly look at MagicLisper in action on a simple problem. Consider the following specification for a function which finds the length of a list:

```
() → 0              [10 steps]
(8) → 1             [20 steps]
(10 4 7 2) → 4      [50 steps]
```

To solve this, MagicLisper first determines the type of the program implied by the specification: in this case, it is a function mapping a list of integers to an integer. It then performs an iterative deepening search through the space of programs matching that type; on the nth iteration, it generates and tests programs whose total weight is less than or equal to n. When testing a program, MagicLisper runs it on each training input in turn, for no more than the specified number of steps in each case. The whole search finishes when MagicLisper finds the program with the smallest weight that satisfies all of the training examples, which is in this case:

```
(λ (a1) (paralist (λ (a2 a3 a4) (inc a4)) a1 0))
```

The above program has a weight of 11.6, so is found on the 12th search iteration.

5. Incremental learning experiments

In this section, we describe a set of experiments with MagicLisper that demonstrate the incremental learning mechanism of section 3, that in which solution programs are successively added to the system's object language as new primitives.

5.1 Method and results

We measured the performance of MagicLisper on four problem sequences, both with and without the aid of incremental learning in each case. Full specifications of these problem sequences along with the experimental results are given in figures 2, 3, 4, and 5. Each specification consists of a

Name	Type	Weight
— The empty list —		
`nil`	`list`	2.1
— List operations —		
`cons`	`(λ (int list) list)`	2.1
`car`	`(λ (list) int)`	3.2
`cdr`	`(λ (list) list)`	3.2
— Integer constants —		
`0`	`int`	3.4
`1`	`int`	3.4
— Integer operations —		
`inc`	`(λ (int) int)`	3.4
`dec`	`(λ (int) int)`	3.4
`+`	`(λ (int int) int)`	3.4
`*`	`(λ (int int) int)`	3.4
— If-then-else —		
`if`	`(λ (bool int int) int)`	2.5
`if`	`(λ (bool list list) list)`	2.5
— Boolean constants —		
`t`	`bool`	3.5
`f`	`bool`	3.5
— Boolean operations —		
`not`	`(λ (bool) bool)`	3.5
`and`	`(λ (bool bool) bool)`	3.5
`or`	`(λ (bool bool) bool)`	3.5
— Integer comparions operations —		
`eql`	`(λ (int int) bool)`	2.0
`<`	`(λ (int int) bool)`	2.0
— Morphisms —		
`paranat`	`(λ ((λ (int int) int) int int) int)`	4.0
`paranat`	`(λ ((λ (int list) list) int list) list)`	4.0
`paranat`	`(λ ((λ (int bool) bool) int bool) bool)`	4.0
`paralist`	`(λ ((λ (int list int) int) list int) int)`	4.0
`paralist`	`(λ ((λ (int list list) list) list list) list)`	4.0
`paralist`	`(λ ((λ (int list bool) bool) list bool) bool)`	4.0
`analist`	`(λ ((λ (list) list) list) list)`	4.5

Table 1. The default library of primitive functions and constants used by MagicLisper. The type system consists of: integers (`int`), lists of integers (`list`), and booleans (`bool`). A compound type expression of the form: `(λ (`a b `...) `r`)` represents a function whose argument types are a, b, etc., and whose return type is r. The role of the weights is to bias the system towards using certain primitives more than others when constructing programs; primitives with lower weights are used more frequently (see section 4.2).

```
(define (paranat f n x)
  (if (zero? n)
      x
      (f (- n 1) (paranat f (- n 1) x))))

(define (paralist f lst x)
  (if (null? lst)
      x
      (f (car lst) (cdr lst) (paralist f (cdr lst) x))))

(define (analist f lst)
  (let ((pair (f lst)))
    (if (null? pair)
        '()
        (cons (car pair) (analist f (cdr pair))))))
```

Figure 1. Definitions of MagicLisper's morphism primitives given in the Scheme dialect of Lisp: *natural number paramorphism*, *list paramorphism*, and *list anamorphism*.

Name	Description	Time / s (custom weights)	Time / s (uniform weights)	Speed-up factor
`append`	Appends two lists together.	< 0.1	14.5	> 145.0
`make-list`	Constructs the list of n instances of a given value.	0.1	13.3	133.0
`length`	Finds the length of a list.	0.2	1.9	9.5
`sum-elems`	Finds the sum of the elements in a list.	0.5	19.9	39.8
`evenp`	Tests if a given integer is even.	0.7	1.7	2.4
`nth`	Finds the nth element of a list.	0.9	26.0	28.9
`last-elem`	Finds the last element of a list.	1.5	248.5	165.7
`member`	Tests if a given value is a member of a list.	6.7	> 251.4	> 37.5
`pow`	Raises one integer to the power of another.	9.6	31.2	3.3

Table 2. Some typical problems that MagicLisper can solve without the aid of incremental learning. In each case, between 3 and 5 training examples were given. Solution times were measured in two different scenarios: 'custom weights', in which the lexical variable and primitive weights were set up as described in section 4.2, and 'uniform weights', in which the lexical variable and primitive weights were all set to the value 1. The 'speed-up factor' column gives the proportional increase in speed due to the custom weights: time (uniform weights) divided by time (custom weights). The measurements were made on a 2 GHz Intel Core II Duo desktop PC with 2 GB of RAM running GNU CLISP.

main problem, and a sequence of sub-problems whose solutions may act as building blocks out of which the solution to the main problem can be constructed. For example, in the `sort` problem sequence (figure 4) we tasked our system with inferring an algorithm to sort a list of numbers. Sub-problems included the simpler but related task of taking the smallest element out of a list and bringing it to the front (`extract-least-elem`), and the yet simpler tasks of finding the smallest element in a list (`least-elem`), and of removing a given element from a list (`remove-elem`).

When designing the problem sequences, we used our knowledge of how one might implement the solution programs by hand in order to choose appropriate sub-problems. We also used some degree of trial and error in tweaking the problem sequences until incremental learning worked effectively (for example, `remove-first-block` was originally the first stage in our design for the `block-lengths` problem sequence; we added an extra stage, `car-p`, when it became apparent that our system was taking too long to solve `remove-first-block` from the default starting conditions). For now, let us emphasise the point that readily comprehensible and effective problem breakdowns often exist. In the next section (6) we shall consider in detail the issue of how much human effort is required to produce these problem breakdowns, as well as what ways can be developed to reduce or remove the need for this human effort.

For every problem and sub-problem, in order to obtain some guarantee that the program found was indeed the correct general solution, we checked it against a set of test examples. When designing our problem specifications, if any

deref-list: *dereferences a list of indices into another list.*

— Training examples —

`(), (7) → ()` [20 steps]

`(0), (6) → (6)` [50 steps]

`(1 0 2), (8 6 4 5) → (6 8 4)` [200 steps]

— Test examples —

`(3 2 2 1 3 4 0 5), (77 42 3 −10 8 61) →`
`(−10 3 3 42 −10 8 77 61)`

`(8 4 7), (9 5 2 5 8 4 1 9 1 7) →`
`(1 8 9)`

Incremental specification

1. **nth**: *returns the nth element of a list.*

   — Training examples —

   `0, (5) → 5` [15 steps]

   `1, (8 6) → 6` [30 steps]

   `3, (4 10 77 34 58) → 34` [150 steps]

   — Test examples —

   `8, (8 4 9 3 7 1 9 2 5 4 7) → 5`

   `4, (11 23 45 15 27 89 102 56) → 27`

2. **deref-list**

Results

Stage	Time / s	Depth	Solution
nth	0.9	12	(λ (a1 a2) (car (paranat (λ (a3 a4) (cdr a4)) a1 a2)))
deref-list	3.6	13	(λ (a1 a2) (paralist (λ (a3 a4 a5) (cons (nth a3 a2) a5)) a1 nil))
Total	**4.5**		

Non-incremental: TIMEOUT (950.2 seconds, depth 18)

Figure 2. The `deref-list` problem sequence: specification and results.

reverse: *reverses a list.*

— Training examples —

`() → ()` [20 steps]

`(8) → (8)` [40 steps]

`(3 7) → (7 3)` [150 steps]

`(9 4 7 1) → (1 7 4 9)` [800 steps]

— Test examples —

`(2 9 1 7 −3 4 8 9 10 12) →`
`(12 10 9 8 4 −3 7 1 9 2)`

`(6 4 5 2 1 1 1 8 2) →`
`(2 8 1 1 1 2 5 4 6)`

Incremental specification

1. **append-elem**: *appends an element to the end of a list.*

   — Training examples —

   `8, () → (8)` [15 steps]

   `4, (9) → (9 4)` [30 steps]

   `7, (3 8 1) → (3 8 1 7)` [100 steps]

   — Test examples —

   `6, (4 7 1 3 9 8 6) → (4 7 1 3 9 8 6 6)`

   `3, (8 8 8 8 8) → (8 8 8 8 8 3)`

2. **reverse**

Results

Stage	Time / s	Depth	Solution
append-elem	1.0	12	(λ (a1 a2) (paralist (λ (a3 a4 a5) (cons a3 a5)) a2 (cons a1 nil)))
reverse	0.1	10	(λ (a1) (paralist (λ (a2 a3 a4) (append-elem a2 a4)) a1 nil))
Total	**1.1**		

Non-incremental: SOLUTION FOUND (569.6 seconds, depth 19)

(λ (a1) (paralist (λ (a2 a3 a4) (paralist (λ (a5 a6 a7) (cons a5 a7)) a4 (cons a2 nil))) a1 nil))

Figure 3. The `reverse` problem sequence: specification and results.

sort: *sorts a list of integers in ascending order.*

— Training examples —

`()` → `()`	[30 steps]
`(7)` → `(7)`	[100 steps]
`(4 2)` → `(2 4)`	[500 steps]
`(9 8 7)` → `(7 8 9)`	[2000 steps]
`(3 2 3 2 3)` → `(2 2 3 3 3)`	[10000 steps]

— Test examples —

`(10 6 −30 7 2 5 −2 3 1 6 4)` →
`(−30 −2 1 2 3 4 5 6 6 7 10)`

`(1 1 1 8 6 8 6 4 3 3 1 1)` →
`(1 1 1 1 1 3 3 4 6 6 8 8)`

`(10 2 105 −78 46 45 23)` →
`(−78 2 10 23 45 46 105)`

Incremental specification

1. **remove-elem**: *removes the first instance of a given element from a list.*

— Training examples —

`6, (6)` → `()`	[15 steps]
`7, (8 7)` → `(8)`	[30 steps]
`3, (3 3)` → `(3)`	[30 steps]
`10, (2 4 10 7 2 1)` → `(2 4 7 2 1)`	[200 steps]

— Test examples —

`43, (9 56 43 2 7)` → `(9 56 2 7)`

`8, (6 8 4 8 2 8)` → `(6 4 8 2 8)`

`9, (7 5 2 9)` → `(7 5 2)`

2. **min**: *returns the smaller of two integers.*

— Training examples —

`2, 1` → `1`	[10 steps]
`3, 10` → `3`	[10 steps]
`7, 7` → `7`	[10 steps]

— Test examples —

`−5, −10` → `−10`

`−3, 20` → `−3`

`27, 27` → `27`

3. **least-elem**: *returns the smallest element in a list of integers.*

— Training examples —

`(3)` → `3`	[20 steps]
`(8 4 7)` → `4`	[100 steps]
`(9 6 2 9 2)` → `2`	[200 steps]

— Test examples —

`(10 7 45 5 7 8)` → `5`

`(77 34 59 34 208)` → `34`

4. **extract-least-elem**: *brings the smallest element to front of a list of integers.*

— Training examples —

`(8)` → `(8)`	[50 steps]
`(10 4)` → `(4 10)`	[200 steps]
`(8 6 2 7 2 5)` → `(2 8 6 7 2 5)`	[2000 steps]

— Test examples —

`(3 2 1 2 3)` → `(1 3 2 2 3)`

`(54 70 14 59 14 20)` → `(14 54 70 59 14 20)`

5. **sort**

Results

Stage	Time / s	Depth	Solution
remove-elem	7.8	14	(λ (a1 a2) (paralist (λ (a3 a4 a5) (if (eql a3 a1) a4 (cons a3 a5))) a2 a2))
min	0.0	7	(λ (a1 a2) (if (< a2 a1) a2 a1))
least-elem	0.7	13	(λ (a1) (paralist (λ (a2 a3 a4) (min a4 a2)) a1 (car a1)))
extract-least-elem	3.3	14	(λ (a1) (cons (least-elem a1) (remove-elem (least-elem a1) a1)))
sort	4.5	14	(λ (a1) (analist (λ (a2) (paralist (λ (a3 a4 a5) (extract-least-elem a5)) a2 a2)) a1))
Total	**16.3**		

Non-incremental: TIMEOUT (586.6 seconds, depth 19)

Figure 4. The `sort` problem sequence: specification and results.

failure occurred at the testing stage, we added new training examples and re-ran the experiment. For the final specifications given in the figures, every solution program has passed all of its test examples.

We recorded the times taken for MagicLisper to solve the stages of each sequence. Total times was determined by adding these values together. Following each sub-problem in a sequence, the inferred solution program was added to the library of primitives and assigned a weight of 2.5, 2.5, 3.5, or 3.0 in the case of problem sequences `deref-list`, `reverse`, `sort`, and `block-lengths` respectively. The library of primitives was reset to its default state between problem sequences. We also tested how our system fared when solving each main problem on its own with the default primitive library, i.e. without incremental learning. We allowed at least 500 seconds for every problem; if this time limit was exceeded then the computation was aborted after allowing for the current search iteration to finish, and 'TIMEOUT' was indicated in the results table. Also given in each results table are the search depths, in units of program weight, at which any solution was found or a timeout occurred, as well as the solution programs themselves. The experiments were performed on a 2 GHz Intel Core II Duo desktop PC with 2 GB of RAM running GNU CLISP.

block-lengths: *replaces all blocks of consecutive identical elements in a list with their lengths.*

— Training examples —

`() → ()`	[50 steps]
`(8) → (1)`	[200 steps]
`(7 6) → (1 1)`	[700 steps]
`(8 8 3 4) → (2 1 1)`	[5000 steps]
`(6 5 5 4) → (1 2 1)`	[5000 steps]

— Test examples —

```
(7 7 7 7 7 5 5 5 7 7 2 2 4 9 9 9) →
   (5 3 2 2 1 3)
(5 8 8 4 9 1 2 1 2) →
   (1 2 1 1 1 1 1 1)
(0 0 0 0 0 7 0 0 0 5 5 5 5 5 5) →
   (5 1 3 6)
```

Incremental specification

1. **car-p**: *tests whether an object is the first element of a list.*

— Training examples —

`0, () → f`	[15 steps]
`1, () → f`	[15 steps]
`4, (4) → t`	[15 steps]
`5, (2) → f`	[15 steps]
`8, (8 2) → t`	[15 steps]
`7, (6 2 7) → f`	[15 steps]

— Test examples —

```
7, (8 7 7 6 4 7) → f
3, (3 8 1 4) → t
```

2. **remove-first-block**: *removes the first block of consecutive identical elements from a list.*

— Training examples —

`(8) → ()`	[30 steps]
`(4 6) → (6)`	[100 steps]
`(1 3 1 3) → (3 1 3)`	[400 steps]
`(9 9 8 6 9 3) → (8 6 9 3)`	[1000 steps]
`(5 5 5 5 4 9) → (4 9)`	[1000 steps]

— Test examples —

```
(7 7 7 7 4 4 3 3 7 8 8 7 2 2) →
   (4 4 3 3 7 8 8 7 2 2)
(1 6 5 1 2 2 2) → (6 5 1 2 2 2)
(9 9 9 9 9) → ()
```

3. **length**: *finds the length of a list.*

— Training examples —

`() → 0`	[10 steps]
`(8) → 1`	[20 steps]
`(10 4 7 2) → 4`	[50 steps]

— Test examples —

```
(8 4 7 3 2 9 1 1 2) → 9
(92 −8 7 83 24) → 5
```

4. **length-first-block**: *finds the length of the first block of consecutive identical elements in a list.*

— Training examples —

`(8) → 1`	[50 steps]
`(4 6) → 1`	[100 steps]
`(9 9 8 6 9 3) → 2`	[1000 steps]

— Test examples —

```
(3 3 3 3 8 7 6 3 4 5) → 4
(5 5 5 5 5 5 5 2 2) → 7
```

5. **convert-first-block-to-length**: *replaces the first block of consecutive identical elements in a list with its length.*

— Training examples —

`() → ()`	[20 steps]
`(8) → (1)`	[100 steps]
`(7 6) → (1 6)`	[400 steps]
`(8 8 3 4) → (2 3 4)`	[2000 steps]
`(5 5 5 3) → (3 3)`	[2000 steps]

— Test examples —

```
(8 8 8 6 6 6 6) → (3 6 6 6 6)
(4 1 5 4 2 2) → (1 1 5 4 2 2)
```

6. **block-lengths**

Results

Stage	Time / s	Depth	Solution
car-p	0.4	11	(λ (a1 a2) (paralist (λ (a3 a4 a5) (eql a3 a1)) a2 f))
remove-first-block	0.5	12	(λ (a1) (paralist (λ (a2 a3 a4) (if (car-p a2 a3) a4 a3)) a1 a1))
length	0.2	12	(λ (a1) (paralist (λ (a2 a3 a4) (inc a4)) a1 0))
length-first-block	6.9	15	(λ (a1) (length (paralist (λ (a2 a3 a4) (cdr a4)) (remove-first-block a1) a1)))
convert-first-block-to-length	4.8	14	(λ (a1) (paralist (λ (a2 a3 a4) (cons (length-first-block a1) (remove-first-block a1))) a1 a1))
block-lengths	0.1	9	(λ (a1) (analist (λ (a2) (convert-first-block-to-length a2)) a1))
Total	**12.9**		

Non-incremental: TIMEOUT (561.8 seconds, depth 19)

Figure 5. The `block-lengths` problem sequence: specification and results.

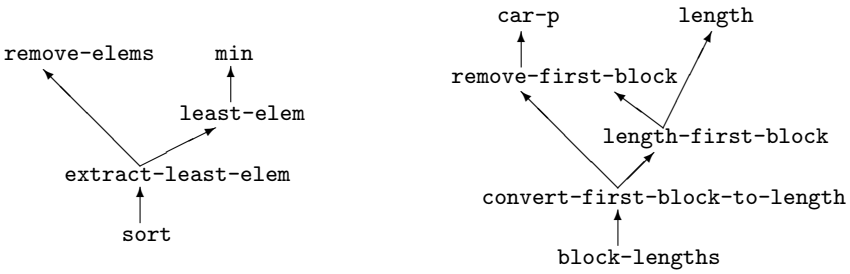

Figure 6. Dependency graphs for the solutions to the `sort` and `block-lengths` problem sequences. Each arrow $x \rightarrow y$ means 'program x invokes program y'.

5.2 Analysis

The total times taken for our system to solve the incremental specifications ranged between 1 and 16 seconds. On the other hand, all of the non-incremental scenarios took more than 500 seconds, with a solution only being found at all in the case of `reverse`. This amounts to an increase in speed due to incremental learning of at least a factor of thirty in every case.

On inspection of the solution programs for the incremental sequences, we see that the majority of programs do indeed invoke earlier solutions, as expected. Indeed, for the longer sequences `sort` and `block-lengths` we can visualise a graph of dependencies between the solution programs (figure 6).

The timing results indicate that, at worst, incremental learning can greatly improve the performance of IP, while, at best, it is able to make otherwise intractable problems tractable. To see why this should be the case, consider the following computational complexity argument. Assuming that it takes constant time to generate and test each program, then the time taken for MagicLisper's search algorithm to solve a given problem will be proportional to the total number of programs generated. We expect this to be approximately $\mathrm{O}[b^d]$, where b is the search branching factor, roughly proportional to the size of the primitive library, and d is the search depth of the lowest-weight solution program that exists for the problem. Now, if we make an assumption that with incremental learning we can always divide a problem into sub-problems whose solution depths are bounded by a constant d_0, then the time taken to solve the problem in incremental stages is no more than $\mathrm{O}[n(b_0+n\Delta b)^{d_0}]$, where n is the number of stages, b_0 is the branching factor of the default primitive library and Δb is the increase in the branching factor that occurs each time we add a new primitive. Let us also assume that the number of stages required to satisfactorily break down a problem is roughly proportional to the depth of the lowest-weight solution program that we'd get if the problem were solved non-incrementally, in other words, $n = kd$. This gives us a time taken of $\mathrm{O}[kd(b_0 + kd\Delta b)^{d_0}]$, or simply $\mathrm{O}[d^{d_0+1}]$ with respect to d, if the problem is solved incrementally, compared with $\mathrm{O}[{b_0}^d]$ if it is solved non-incrementally. In this way, IP with incremental learning can allow a system solve, in polynomial time, problems that take exponential time with non-incremental IP.

6. Limitations and further work

The main contribution of this paper has been to demonstrate a simple, working methodology for incremental learning in IP. This methodology involved equipping a brute-force search based IP system with an ability to reuse solution programs by adding them to its primitive library. We showed that this mechanism can be effective by demonstrating its use on four problems, each of which had been broken down into an appropriate sequence of sub-problems. Our IP system was able to solve the problems orders of magnitude more quickly when making using of the incremental sequences than when simply solving the main problems in isolation.

In this section we address the limitations of our simple incremental learning methodology; in particular we talk about the difficulties involved in constructing problem sequences. We consider how to overcome these limitations, and discuss how, by eliminating the need for problem sequences to be designed by a human expert, we aim to enable a much more useful, autonomous form of incremental learning.

6.1 Limitations of the simple methodology

The main drawback of the simple incremental learning methodology presented in this paper is the significant amount of human effort and expertise required to design effective problem sequences. Based on our experience designing problem sequences for MagicLisper, we feel that the need for this effort and expertise is largely due to what we shall call 'brittleness' in the system's learning mechanism. In other words, problem specifications must obey certain conditions in order for learning to work, and they 'break easily', i.e. if these conditions are not met perfectly, then the system will fail to find a solution at all.

One source of brittleness in our mechanism is the fact that solutions to sub-problems are only useful if a solution to the main problem can be expressed in terms of them directly. It is not enough for a sub-problem simply to be related to the main problem, for example if their solutions would share some common structure. In consequence, the success

or failure of incremental learning is very sensitive to the exact choice and order of sub-problems. Often, the only way to predict if a particular sub-problem will be effective is to use ones knowledge of how one might implement the target program by hand; in other words, using the IP system does not save one much effort over hand-coding the program. Our methodology suffers from brittleness in two other ways too. Firstly, the IP system will not tolerate any error or noise in the training examples. Secondly, if any of the step counts associated with the training examples are too low, the system will again completely fail to find a solution.

6.2 Overcoming the limitations

Our simple methodology seems capable of scaling up to relatively complex problems, but at the cost of a degree of human effort expended in designing problem sequences at least as great as would be required to code the solutions by hand. In this subsection we discuss how to eliminate the three sources of 'brittleness' described in the last subsection, with the aim of developing a mechanism that is flexible enough to perform incremental learning over loosely constructed sets of problems, rather than precisely constructed sequences.

The need to specifiy step counts with training examples should be the easiest limitation to overcome. In MagicHaskeller, it is already unnecessary to specify step counts, because the system simply tests all programs until termination, relying on the fact the primitive library belies the possibility of infinite loops. However, we don't expect this approach to remain feasible as we start to generate more complex programs, because the number of programs that run for a long time before termination will become much larger. Instead, we propose using an algorithm like 'Levin search' (Schmidhuber 2004), in which the iterative deepening nature of our IP search is extended so as to automatically re-test programs for longer and longer step counts as the search progresses.

The need for a solution to a main problem to be expressible directly in terms of solutions to sub-problems could be overcome as follows. Suppose that we modify our incremental learning mechanism such that, instead of adding actual solution programs to the primitive library, it attempts to derive re-usable procedural abstractions from groups of solution programs, and then adds these abstractions as the new primitives. The potential re-usabilility of a procedural abstraction can be measured objectively using a principle of 'minimum description length': if a procedure, when reused in multiple solution programs, serves to reduce the combined size of these programs by more than its own size, then we can deem it a useful abstraction. Though the best way to discover candidate abstractions is an open question, it would seem a reasonable starting point to try a brute-force search. This method of incremental learning would be much more adaptable and generic than our original mechanism, in that it should be able to extract useful inductive bias from almost any kind of shared structure or commonality between solution programs. We know of at least one previous implementation of a similar idea: the 'Duce' system (Muggleton 1987) can discover abstractions that encapsulate shared structure among groups of statements in propositional logic.

To overcome the lack of toleration of errors or noise in the training examples, we feel that the most satisfactory solution will ultimately be to reformulate our IP methodology within a probabilistic framework. In such a framework, a program would no longer describe a deterministic mapping from inputs to outputs, rather it would represent a conditional probability distribution over the set of possible outputs given the inputs. Such a reformulation is highly desirable if our aim is to develop a machine learning technique of practical use, since real-world data is usually noisy. Indeed, the development of probabilistic frameworks for IP is an active area of research, particularly within inductive logic programming (De Raedt and Kersting 2004).

In overcoming the above limitations, our eventual goal is to produce an IP methodology capable of performing incremental learning simply from a corpus of data, without the need for that data to be organised into problem sequences by a human expert. To see how this might work, first consider how a system could perform incremental learning if provided with a large bank of related problems of various difficulties, in no particular order. Such a system could repeatedly scan through the problems, briefly attempting to solve each as it goes. Some of the problems might be easy enough to solve immediately, and the system could then use the solutions of these to derive procedural abstractions which it would add to its primitive library. On the next scan through the problem bank, these new primitives should enable the system to solve some problems that were previously out of its reach. Ideally, the process iterates until most of the problems are solved. Consider next how one might extend this idea in order to create a system capable of automonomously learning a model for a complex environment or corpus of data. In a such a situation, it might often be the case that various parts of the environment or corpus can be described by simple models. By analogy with the 'bank of problems' scenario, one may imagine an incremental learning system that initially looks for these simple models, adds abstractions derived from those models to its background knowledge, then searches for more complex models, and so on. We may think of this process as an automation of the scientific method.

7. Conclusion

In this paper, we have demonstrated a simple but effective incremental learning mechanism for an inductive programming system. It works by having the system incorporate solution programs into its object language as new primitive functions as it progresses through a sequence of problems. The mechanism is capable of producing orders of magnitude improvements in problem solving performance, but at the expense of considerable human effort spent in designing appropriate problem sequences. However, we have sketched

a number of possible improvements to the mechanism which should reduce or remove much of the need for this human guidance. Our aim is that this methodology can eventually be developed into a powerful generic machine learning technique by which a system can learn a model of a large, complex dataset in an autonomous fashion.

Acknowledgments

Thank you to my MSc supervisor Michael O'Boyle, for his support and encouragement on this project.

References

L. Augusteijn. Sorting morphisms. In *3rd International Summer School on Advanced Functional Programming*, volume 1608 of *LNCS*, pages 1–27. Springer-Verlag, 1998.

L. De Raedt and K. Kersting. Probabilistic inductive logic programming. In *ALT*, volume 3244 of *LNCS*, pages 19–36. Springer, 2004.

M. Hofmann, E. Kitzelmann, and U. Schmid. A unifying framework for analysis and evaluation of inductive programming systems. In *Proceedings of the Second Conference on Artificial General Intelligence (AGI-09)*, 2009.

S. Katayama. Systematic search for lambda expressions. In *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005*, volume 6, pages 111–126. Intellect, 2007.

K. Khan, S. Muggleton, and R. Parson. Repeat learning using predicate invention. In *Inductive Logic Programming*, volume 1446 of *LNCS*, pages 165–174. Springer, 1998.

E. Kitzelmann. Data-driven induction of recursive functions from input/output-examples. In *Proceedings of the ECML/PKDD 2007 Workshop on Approaches and Applications of Inductive Programming (AAIP'07)*, pages 15–26, 2007.

S. Muggleton. Duce, an oracle based approach to constructive induction. In *IJCAI-87*, pages 287–292, 1987.

J. R. Olsson. Inductive functional programming using incremental program transformation. *Artificial Intelligence*, 74(1):55 – 83, 1995.

J. R. Quinlan and R. M. Cameron-Jones. FOIL: A midterm report. In *Proceedings of the 6th European Conference on Machine Learning*, LNCS, pages 3–20. Springer-Verlag, 1993.

J. Schmidhuber. Optimal ordered problem solver. *Machine Learning*, 54(3):211–254, 2004.

J. Schmidhuber, J. Zhao, and M. Wiering. Shifting inductive bias with success-story algorithm, adaptive levin search, and incremental self-improvement. *Machine Learning*, 28(1):105–130, 1997.

R. J. Solomonoff. Progress in incremental machine learning. Given at *NIPS Workshop on Universal Learning Algorithms and Optimal Search, Dec. 14, 2002, Whistler, B.C., Canada.*, 2002. URL `http://world.std.com/~rjs/pubs.html`.