ReDuce: Linear-time Inductive Compression using Greedy Folding

S.H. Muggleton

Nanjing University and Imperial College London
s.muggleton@gmail.com, s.muggleton@imperial.ac.uk

Abstract. This paper describes a new *Inductive Logic Programming system*, *ReDuce*, which is a linear time variant of a key element of the author's *DeepLog* system. *ReDuce* differs from *DeepLog* by replacing the Meta-Interpretive Learning hypothesis generator by a novel inductive variant of an existing near-optimal grammar-based greedy text compression algorithm. Having identified minimal length input/output certificate sequences for the examples *ReDuce* iteratively applies *star* (repeat) and *fold* operations in order to generalise the examples and invent referenced sub-predicates. This compaction process is guaranteed by construction to converge, and results in a deeply structured recursive program which both compacts and generalises the examples. The approach is proved to run in time and space which are linear with respect to the certificate sequence lengths. The identified regular expressions are used to generate an H^2_2 logic program from the learned folded certificates. Experiments indicate hypothesis construction time to be, in some case, more than ten-thousand-fold faster than corresponding examples in the author's previously published *DeepLog* paper. As with *DeepLog*, an existing Bayesian sample complexity result shows low generality hypotheses can be learned with high expected accuracy from small numbers of positive examples. In further work the author aims to explore identification of noisy examples using a modified text compression approach.

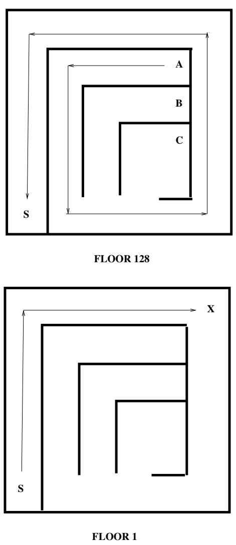
Keywords: Inductive Programming · Text Compression · Grammar.

1 Introduction

Imagine trying to devise a general fire escape program for evacuating a building. The program must ensure everyone in a bedroom exits the building via the shortest path. DeepLog [7] identifies a program, involving multiple recursions and introduction of new predicates (see Figure 1), from a single example sequence (or certificate). However, since search time increases non-linearly with sequence length, DeepLog only identifies programs for buildings of at most 16 storeys high.

Question: Is it possible, in time linear in certificate sequence length, to identify a compact structured program?

a) Floorplan



b) Hypothesised program (23 clauses):

```
f128(X,Y) :- defn_1(X,Z),f128_1(Z,Y).
f128_1(X,Y) :- star_e(X,Z),f128_2(Z,Y).
f128_2(X,Y) :- star_n(X,Z),f128_3(Z,Y).
f128_3(X,Y) :- defn_1(X,Z), f128_4(Z,Y).
f128_4(X,Y) :- star_d(X,Z), f128_5(Z,Y).
f128_5(X,Y) :- star_n(X,Z), star_e(Z,Y).

defn_1(X,Y) :- star_w(X,Z),star_s(Z,Y).

star_d(X,Y) :- eq(X,Z), eq(Z,Y).
star_d(X,Y) :- d(X,Z), star_d(Z,Y).
star_e(X,Y) :- eq(X,Z), eq(Z,Y).
star_e(X,Y) :- e(X,Z),star_e(Z,Y).
..
d([d|T],T).
e([e|T],T).
..
eq(X,X).
```

c) Certificate sequence (185 steps):

$\langle w, w, .., s, s, .., e, e, ..n, n, .., w, w, .., s, s, .., d, d, .., n, n, .., e, e, ..\rangle$

d) Timings

Floors	16	128	1024
DeepLog	6.3s	-	-
ReDuce	0.0005s	0.0005s	0.0008s
Ratio	12,600	-	-

Fig. 1. a) Floorplan 2-128 have identical structure, "S" is Stair, "X" is Exit, A-C are starting bedroom positions. b) The hypothesised program is generated from c) the certificate sequence of 185 step actions from {d,e,n,s,w}. d) Hypothesis generation timings with 16, 128 and 1024 floors. DeepLog fails with more than 16 floors.

Figure 1[1] shows the layout of the top and bottom floors of a building. An example certificate is provided which indicates a minimal escape sequence consisting of 185 individual actions {d,e,n,s,w} from position A on Floor 128 to the Exit position X on Floor 1. Given this sequence ReDuce[2] hypothesises a general hierarchical program which generates a minimal path sequence from any of the bedroom positions A,B,C on floors 2-128 to the exit position X. The tim-

[1] Timings in Figure 1d) for DeepLog and ReDuce were both based on SWI Prolog runnning on a single core 11th Gen Intel(R) i7-1165G7 @ 2.80GHz.

[2] *ReDuce* is named after the author's earlier *Duce* system [4] (see Section 2.1) which uses information reduction to guide a greedy search for hypotheses.

ings (Figure 1d) indicate hypothesis generation for *ReDuce* strongly outperforms *DeepLog* and increases at most linearly with building height.

The paper is organised as follows. Related work is described in Section 2. In Section 3 the mathematical framework for the ReDuce algorithm is introduced. Section 4 describes the Reduce algorithm alongside a time-complexity analysis of the associated algorithms for 1) identifying high frequency pairs and 2) interleaving identification of repeating sequences and greedy folding. Section 5 describes experiments comparing the comparative performance of ReDuce and DeepLog on various example sets. Section 6 concludes and describes potential further work.

2 Related work

2.1 Propositional structure learning by compression

Duce [4] is a Machine Learning system which introduces propositional domain features on the basis of a set of examples. Duce uses six transformation operators to successively compress the examples by generalisation and feature introduction. Duce's main achievement was the restructuring of a substantial expert system for deciding whether positions within the chess endgame of King-and-Pawn-on-a7 versus King-and-Rook (KPa7KR) are won-for-white or not.

The ReDuce system described in this paper is similar to Duce in its use of successive compressive operations to introduce a structured hierarchy. By contrast, the ReDuce system constructs first-order recursive logic programs, allowing predicate arguments to carry and revise the state (eg the agent's position in the fire example) during execution of the program.

2.2 Grammar Induction

The earliest theoretical framework for learning formal grammars from examples was published in 1967 by Gold [2], and is known as *Identification in the Limit*. The framework assumes learning algorithms are provided with a finite enumeration of both hypotheses and examples of a target language, for identifying consistent hypotheses. Gold proved that, in general, for infinite formal languages, such as regular and context-free languages in the Chomsky hierarchy, identification in the limit is not possible when learning from positive examples alone. The reason is that for such language classes, given a finite number of examples, at no point can an algorithm discriminate between the most general language, consisting of all possible sequences from a given alphabet, and the most specific language, containing only the examples provided so far. Gold [2] pointed out the apparent disparity between this formal result and existing psycholinguistic studies by McNeill [3], which indicated that children learn language largely from positive examples, and tend to ignore corrections to their use of grammar.

2.3 Bayesian positive-only learning

In 1996 [6], the author introduced a Bayesian framework to analyse the learning of logic programs from positive-only examples. A Bayesian prior over the hypothesis space is assumed. It was shown that polynomial time logic programs can be learned with high accuracy from a randomly selected positive example sequence. This result goes beyond the positive-only results of Gold, and supports efficient learning of infinite languages and programs, since the class of Logic Programs is equivalent to Chomsky Type 0 unrestricted grammars. However, [6] falls short of providing effective error bounds for one-shot learning.

2.4 Bayesian one-shot learning

In [7] the author introduced a framework for analysing Expected Error (EE) when learning from one example. The approach is based on an adaptation of the author's existing Bayesian analysis of positive-only learning [6]. The approach enabled an upper-bound error analysis of the human phenomenon of one-shot learning. The EE bounds for one-shot learning were shown to be a special case of Bayesian positive-only learning in which the target theory has generality below 0.01. The paper introduced an implementation called Deeplog and demonstrated experimentally that the EE bound holds on a variety of problems, including two grammar learning problems and a general program learning problem involved in reversing a list. Deeplog was implemented as a Meta-interpretive Learning system.

2.5 Meta-Interpretive Learning

The present paper builds on earlier work in Meta-Interpretive Learning (MIL) [8, 9], which is a sub-category of Inductive Logic Programming (ILP) [5, 10]. MIL learners attempt to prove a set of goals by repeatedly fetching higher-order metarules whose heads unify with a given goal. By contrast, the approach described in this paper replaces a potentially expensive search process by a greedy algorithm which interlaces Regular Expression star-repeats and greedy folding (see Section 3) to rapidly identify compact hypothesised programs. Compared with an MIL system like DeepLog [7] this approach avoids both a) time-consuming hypothesis construction and b) testing of positive examples in MIL systems since both of these are guaranteed to be consistent by construction.

2.6 Grammar-based Compression

Within Computer Science grammar-based lossless compression algorithms compress text using a context-free grammar (CFG). For instance, the widely used Sequitur algorithm [11] iteratively translates an input text into a Chomsky normal-form grammar by replacing pairs of symbols in the text by symbols associated with a definition. The process produces a structured hierarchical grammar which can be unfolded to regenerate the original text. Neville-Manning and Witten [11] provide a proof that the Sequitur algorithm runs in linear time and space. However, recent interest in the *Smallest Grammar Problem* (SGP) has shown that optimal compression using a Context-Free grammar is NP-hard for unbounded alphabets and NP-complete for bounded alphabets [1].

Going beyond lossless compression, the ReDuce algorithm in this paper interleaves the identification of Regular Expression star-repeats with a greedy variant of the Sequitar approach to folding high-frequency subsequences. This results in a regular grammar which inductively generalises a set of example sequences. The Neville-Manning and Witten proof is extended in Section 4 to show that the Reduce algorithm also runs in linear time and space. ReDuce translates the resulting grammar into an H^2_2 logic program[3] (see Figure 1b) which represents the set of exit sequences as a regular grammar.

3 ReDuce framework

3.1 Lossless compression

For the purposes of this paper we define *lossless compression* as follows.

Definition 1. Lossless compression *Let σ represent a finite set of symbols, σ^* be the countable set of sequences of symbols from σ, $|s|$ be the length of a sequence s and κ_1^{-1} represent a function and its inverse over σ^*. We say that κ_1 is a* lossless compression *function iff $\forall s \in \sigma^*, t \in \sigma^*$ it is the case that $t = \kappa_1(s)$ iff both $s = \kappa_1^{-1}(t)$ and $|t| < |s|$.*

Note that *lossless compression* can be used to reduce the length of a sequence (or text), and that the original sequence can be retrieved without loss of information. Sequitur [11] (see Section 2.6) is a linear-time Lossless compression algorithm.

Lossless compression is usually contrasted with *lossy compression*, in which an approximation of the original text is retrieved.

3.2 Inductive compression

In this paper we consider a variant of lossy compression which is referred to as *inductive compression* since it can be viewed as a form of inductive inference.

Definition 2. Inductive compression *Let σ, σ^* and $|s|$ be as in Definition 1. Additionally let set s be such that $s \in \sigma^*$ and $\kappa_2(s)$ is an encoding of s. We say that $\kappa_2(s)$ is an* inductive compression *of s iff $|\kappa_2(s)| < |s|$.*

Example 1. Both Duce [4] and the ReDuce approach described in this paper are Inductive compression algorithms.

[3] H^2_2 logic programs in this paper consist of clauses of the form $P(x, y) \leftarrow Q(z, y), R(z, y)$ where P, Q, R represent predicate symbols, x, y, z represent universally quantified variables and ← is a right-to-left implication.

We now show that both Lossless compression and Inductive compression algorithms are finitely terminating.

Theorem 1. Finite termination *Assume A is either a Lossless or Inductive Compression algorithm, $s_0 \in \sigma^*$ is a finite sequence of length $|s_0|$. It follows that $s = A(s_0)$ finitely terminates after n applications of κ where $0 \leq n < |s_0|$.*

Proof. Assume Theorem 1 is false. In the case $\kappa(s_0)$ fails, then $s_0 = A(s_0)$ finitely terminates with $n = 0$. So assume $\kappa(s_0)$ succeeds. In this case $s = A(s_0)$ finitely terminates after n applications of κ where $n \geq |s_0|$. However, each application of κ reduces the sequence by at least one, so it follows that $n < |s_0|$, which contradicts the assumption and completes the proof.

3.3 Greedy inductive compression

An algorithm is referred to as *greedy* when locally optimal choices are selected at each stage. In the case of iterative sequence compression this does not guarantee a globally smallest solution, but efficiently produces a near-smallest solution. The ReDuce system described in the next section uses a greedy strategy which iteratively computes the frequency of neighbouring pairs of symbols in a given sequence S. In the case that a subsequence xy has maximum frequency in S of at least 2, then this pair is replaced throughout the sequence by a unique symbol, such as T, which is added as a definition $T \rightarrow xy$. In this paper we refer to this process as sequence folding. Example 2 shows how iterative greedy sequence folding works in lossless compression.

Example 2. Iterative greedy sequence folding. Consider the sequence $S \rightarrow aaabbbbbcccaabbcc$. The frequencies f of pairs of symbols in S are $f(aa) = 3, f(ab) = 2, f(bb) = 4, f(bc) = 1, f(cc) = 3$. A greedy choice for folding is $S \rightarrow aaaTTcccaaTcc, T \rightarrow bb$. When this folding process is iterated it terminates with $S \rightarrow UaTWcUW, T \rightarrow bb, U \rightarrow aa, V \rightarrow cc, W \rightarrow TV$.

3.4 Greedy inductive compression

Example 2 shows how folding supports introduction of nested structure, in the form of sub-expressions. Example 3 below shows how star-repeat expressions, used in Regular Expressions, can generalise sequences by replacing repeated symbols, such as aaa, by star-repeat expressions such as a^*.

Example 3. Inductive compression. Once more consider the sequence $S \rightarrow aaabbbbbcccaabbcc$. Using star-repeat expressions gives $S \rightarrow UVWUVW, U \rightarrow a^*, V \rightarrow b^*, W \rightarrow c^*$, which is an inductive generalisation of the sequence. Following this a greedy lossless compression step introduces further structure, giving $S \rightarrow XWXW, U \rightarrow a^*, V \rightarrow b^*, W \rightarrow c^*, X \rightarrow UV$. In a further lossless iteration this becomes $S \rightarrow YY, U \rightarrow a^*, V \rightarrow b^*, W \rightarrow c^*, X \rightarrow UV, Y \rightarrow XW$. Finally, a star-repeat generalisation produces $S \rightarrow Z, U \rightarrow a^*, V \rightarrow b^*, W \rightarrow c^*, X \rightarrow UV, Y \rightarrow XW, Z \rightarrow Y^*$. Note this grammar has multiple levels of both structure and iteration and cannot be compacted further.

4 Implementation

Below is a top-level pseudo-code description of the ReDuce algorithm.

Algorithm 1 ReDuce(InitialGrammar)

```
1: $G \leftarrow$ StarReplace(InitialGrammar)
2: while Reducible(G) do
3:     Pair in $G \leftarrow$ MaxFrequencyPairInBody(G)
4:     $G \leftarrow$ DefineAndSubstituteAll(Pair,G)
5:     G $\leftarrow$ StarReplace(G)
6: end while
7: return(G)
```

Algorithm 1 can be illustrated by considering Example 3 (Section 3.4). The Initial Grammar (step 1) is $S \rightarrow aaabbbbbcccaabbcc$. This is reduced by Star-Replacement to $S \rightarrow UVWUVW$. This is Reducible (step 2) since both UV and VW have a frequency of at least 2. UV is then selected (step 3) and substituted (step 4) to give $S \rightarrow XWXW$. Star-Replacement has no effect (step 5) since no adjacent symbols are identical. After iterating (step 2) $XWXW$ is reducible since XW has frequency 2, and so (step 4) these are replaced to give $S \rightarrow YY$. On the following iteration this is reduced to $S \rightarrow Z$. This is no longer reducible and so (step 7) the revised grammar is returned together with all introduced definitions[4].

Theorem 2. Correctness of Algorithm 1 *By construction the initial sequence can be generated from the S rule in the grammar G by successive unfolding of symbols in the right-hand side.*

Proof. *Consider Algorithm 1. Let S_0 represent the initial grammar, $S_n = ReDuce(S_0)$ and $S_0, S_1, .., S_n$ represent the sequence of folds leading from S_0 to S_n. By construction there exists a sequence of unfold operations $S_n, S_n - 1, .., S_0$ which generate S_0 from S_n. This completes the proof.*

A critical motivation for this paper is the potential reduction in time-complexity required for hypothesis formation within ILP. We now address the time complexity of Algorithm 1.

Theorem 3. Time complexity *All pairs of symbols and their frequencies can be identified in time linear in the initial sequence.*

[4] Note that Prolog definitions such as those in Figure 1 are generated from grammar rewrite rules such as $P \rightarrow QR$ using meta-rule templates such as $P(x, y) \leftarrow Q(x, z), R(z, y)$. The resulting Prolog rules contain variables, such as x, y, z, which allow State Transformations in the resulting program, enabling the learning of recursively enumerable functions such as *Reverse Uppercase* (see Problem 3 in Figure 4 below.

Proof. Each cycle of Algorithm 1 reduces the right-hand-side of S by at least one symbol. Each such reduction requires constant time when conducted using hash-indexed substitutions. Thus the number of cycles is bounded by the length n of the right-hand-side of S, and there are at most n indexed foldings. This implies that Algorithm 1 terminates in $O(n)$ time and space. This completes the proof.

Note this is the same order of complexity as Neville-Manning and Witten [11] result for the Sequitur algorithm, which is widely used for large-scale text compression.

DeepLog (6.3s)	**ReDuce (0.0008s)**
abc4(X,Y) :- a(X,Z), abc4_1(Z,Y).	abc4(X,Y) :- a(X,Z), abc4_1(Z,Y).
abc4_1(X,Y) :- b(X,Z), abc4_2(Z,Y).	abc4_1(X,Y) :- b(X,Z), abc4_2(Z,Y).
abc4_2(X,Y) :- g(X,Z), h(Z,Y).	abc4_2(X,Y) :- star_3(X,Z), abc4_3(Z,Y).
abc4_2(X,Y) :- cdef(X,Z), abc4_2(Z,Y).	abc4_3(X,Y) :- g(X,Z), h(Z,Y).
cdef(X,Y) :- cd(X,Z),ef(Z,Y).	defn_1(X,Y) :- c(X,Z), d(Z,Y).
cd(X,Y) :- c(X,Z),d(Z,Y).	defn_2(X,Y) :- defn_1(X,Z), e(Z,Y).
ef(X,Y) :- e(X,Z),f(Z,Y).	defn_3(X,Y) :- defn_2(X,Z), f(Z,Y).
	star_3(X,Y) :- eq(X,Z), eq(Z,Y).
	star_3(X,Y) :- defn_3(X,Z), star_3(Z,Y).

Fig. 2. Problem 1: The regular grammars generated by DeepLog and ReDuce given the example $\langle a, b, c, d, e, f, c, d, e, f, g, h\rangle$, with automatically primitive predicates $a, , h$ left out (see Fig. 1). Both solutions represent the grammar ab(cdef)*gh. A special purpose mechanism automatically generates the DeepLog solution, while ReDuce generates equivalent definitions as part of the greedy sequence folding process.

5 Experiments

In this section we compare the speed performance[5] and output theories of ReDuce and DeepLog on the three examples studied in [7][6].

5.1 Problem 1: Regular Grammar

Consider a formal language which involves repeated letter sequences. For instance, the positive example sequence e^+ = "abcdefcdefgh" might be used to exemplify the target language G = ab(cdef)*gh. G corresponds to letter sequences, such as e^+, which have a prefix $\langle a, b\rangle$, a suffix $\langle g, h\rangle$ and zero or more repetitions of $\langle c, d, e, f\rangle$ between them.

[5] Comparitive timings are based on SWI-Prolog running on a single core 11th Gen Intel(R) i7-1165G7 @ 2.80GHz.

[6] Code and example sets at https://github.com/StephenMuggleton/ReDuce and https://github.com/StephenMuggleton/DeepLog .

DeepLog (6.4s)	**ReDuce (0.0005s)**
fire16(X,Y) :- fws(X,Z), fire16_1(Z,Y).	fire16(X,Y) :- defn_1(X,Z), fire16_1(Z,Y).
fire16_1(X,Y) :- fss(X,Z), fire16_1_1(Z,Y).	fire16_1(X,Y) :- star_e(X,Z), fire16_2(Z,Y).
fire16_2(X,Y) :- fns(X,Z), fes(Z,Y).	fire16_2(X,Y) :- star_n(X,Z), fire_3(Z,Y).
fire16_2(X,Y) :- fd(X,Z), fire16_2(Z,Y).	fire16_3(X,Y) :- defn_1(X,Z), fire16_4(Z,Y).
fire16_2(X,Y) :- fes(X,Z), fire16_3(Z,Y).	fire16_4(X,Y) :- star_d(X,Z), fire16_5(Z,Y).
fire16_3(X,Y) :- fns(X,Z), fire16(Z,Y).	fire16_5(X,Y) :- star_n(X,Z), star_e(Z,Y).
	defn_1(X,Y) :- star_w(X,Z), star_s(Z,Y).
	star_d(X,Y) :- eq(X,Z), eq(Z,Y).
	star_d(X,Y) :- d(X,Z), star_d(Z,Y).
	star_e(X,Y) :- eq(X,Z), eq(Z,Y).
	star_e(X,Y) :- e(X,Z), star_e(Z,Y).
	star_n(X,Y) :- eq(X,Z), eq(Z,Y).
	star_n(X,Y) :- n(X,Z), star_n(Z,Y).
	star_s(X,Y) :- eq(X,Z), eq(Z,Y).
	star_s(X,Y) :- s(X,Z), star_s(Z,Y).
	star_w(X,Y) :- eq(X,Z), eq(Z,Y).
	star_w(X,Y) :- w(X,Z), star_w(Z,Y).

Fig. 3. Problem 2: The 16 storey fire exit strategies of DeepLog and ReDuce. Both solutions represent the same solution, though the DeepLog primitive library provides auxiliary predicates such as *fws, fss, fns, fes*. By contrast, reduce creates these automatically as part of the greedy sequence folding process.

5.2 Comparison of DeepLog and ReDuce on Problem 1

Figure 2 shows solutions and timings for DeepLog and ReDuce on Problem 1. Note that ReDuce generates the hypothesis 700 times faster than DeepLog and does not require a specialised mechanism for identifying auxiliary predicates, such as *cdef*.

5.3 Problem 2: Fire escape plan

This problem involves a general fire escape plan for leaving a 16 storey building (see Figure 1). All floors have the same layout as Floor 16, except Floor 1, which contains the EXIT.

5.4 Comparison of DeepLog and ReDuce on Problem 2

Figure 3 shows solutions and timings for DeepLog and ReDuce on Problem 2. Note that ReDuce generates the hypothesis more than 12,000 times faster than DeepLog and, unlike DeepLog, ReDuce uses the greedy sequence folding mechanism to build its own auxiliary predicates for repeatedly moving north, southeast and west, rather than depending on these being available as auxiliary predicates.

DeepLog (0.6s)	**ReDuce (0.0004s)**
revupc(X,Y) :- call1(X,Z), revupc_1(Z,Y).	revupc(X,Y) :- call1(X,Z), revupc_1(Z,Y).
revupc_1(X,Y) :- pop(X,Z), revupc_2(Z,Y).	revupc_1(X,Y) :- star_2(X,Z), return1(Z,Y).
revupc_2(X,Y) :- upcase(X,Z), revupc_3(Z,Y).	defn_1(X,Y) :- pop(X,Z), upcase(Z,Y).
revupc_3(X,Y) :- push(X,Z), return1(Z,Y).	defn_2(X,Y) :- defn_1(X,Z), push(Z,Y).
revupc_3(X,Y) :- push(X,Z), revupc_1(Z,Y).	star_2(X,Y) :- eq(X,Z), eq(Z,Y).
	star_2(X,Y) :- defn_2(X,Z), star_2(Z,Y).

Fig. 4. Problem 3: The Reverse Uppercase programs of DeepLog and ReDuce. Once more both programs represent the same solutions, in this case using the same auxiliary predicates.

5.5 Problem 3: Reverse uppercase function

This problem involves the construction of a function for reversing a sequence of letters while turning them into uppercase (eg *alice* → *ECILA*). Efficient Prolog programs normally require the introduction of an arity 3 auxiliary predicate to reverse a list. The same end is achieved with arity 2 library primitives for DeepLog by dynamically creating a stack in the program state. Since Problem 3 requires such a stack to be of unbounded size, the resulting program can be thought of as a push-down automaton (PDA) as opposed to the finite state automata (FSA) solutions in Problem 1 and Problem 2. PDAs represent a higher expressivity function class than FSAs since the stack can be treated as a Turing-machine tape (see Expressivity discussion in Section 6.2).

5.6 Comparison of DeepLog and ReDuce on Problem 3

Figure 4 shows the solutions and timings for DeepLog and ReDuce on Problem 3. ReDuce generates the same hypothesis more than 1500 times faster than DeepLog using the same auxiliary predicates.

6 Conclusions and Further Work

6.1 Conclusions

This paper describes a new *Inductive Logic Programming system ReDuce*. In Section 1 a motivating example is provided involving the gereration of a strategy to allow optimally efficient fire escape routes from bedrooms in a high-rise building. However, owing to non-linear time and space complexity, DeepLog is unable to find a correct program for building designs higher than 16 storeys. By contrast, the new ReDuce system outperforms DeepLog on this problem by a factor of 12,600, and easily scales to at least 1024 storeys.

Section 2 compares the Meta-Interpretive hypothesis generation approach of DeepLog to the grammar-based compression approach employed by ReDuce. The ReDuce framework is introduced in Section 3, which provides and contrasts definitions of grammar-based *lossless compression* versus a new form of grammar-based *inductive compression*. Lossless and Inductive compression algorithms are shown to terminate and a simple example is provided to contrast the outcome of greedy lossless compression versus greedy inductive compression.

The ReDuce algorithm is described in Section 4. Both the correctness and linear-time complexity are shown. In Section 5 the three problems from [7], used to exemplify DeepLog, are revisited. In all cases ReDuce is at least 1,000 times faster in hypothesising an equivalent Prolog program to that found by DeepLog. Moreover, in Problems 1 and 2 ReDuce introduced predicates which were provided to DeepLog either as user-defined background knowledge or by special purpose mechanisms.

6.2 Further Work

Bayes' Model versus Smallest Grammar The DeepLog paper [7] was motivated by demonstrating that programs can be learned from a small number of positive-only examples. The Bayes' model developed in that paper showed that learning can be achieved, with high probability, from a single example in the case that both a) the generality and b) the size of the target program are low. Problems 1, 2 and 3 in Section 3 have these properties.

The implementation of Algorithm 1 in Section 4 is presented alongside formal proofs of Correctness and Time Complexity. Further formal optimality analysis along the lines of [1] would be required to identify how close learned programs are to a smallest grammar. However, when learning from more than one example, any such analysis should be combined with sample complexity bounds related to those associated with DeepLog [7]. The Bayes' model bounds in DeepLog indicate that error is not minimised by identifying the smallest grammar H, but rather by minimising $sz(H)+m\,(ln\ g(H))$, where $sz(H) = -ln(D_{\mathcal{H}}(H))$ represents the size of the hypothesis. $g(H)$ is the generality of H where $0 \leq g(H) \leq 1$ and $g(H)$ is the sum of the probability of the instances of H.

Further work is required to show how close the greedy approach is to learning a Bayes' predictor with minimal expected error.

Expressivity of learned programs Section 5 describes experiments on three problems. Problem 1 (abc4), Problem 2 (fire16) and Problem 3 (revupc) can each be viewed either as learning a finite automaton or a Chomsky Type-3 regular grammars based on the sequence of the primitive actions provided. However, Problems 1 and 2 simply involve sequentially reading the input tape, while Problem 3 involves a) reading the input tape, b) creating and operating an internal tape representing a stack and c) writing to the output tape. In this way, Problem 3 can be most easily characterised as either a pushdown automaton or a Universal Turing machine program.

Further work is required to identify primitives, other than stack operations on an unbounded tape, which might support learning Type-0 grammars, equivalent to Recursively Enumerable functions and Universal Turing Machine programs.

Identification of noise It should be noted that all the learned programs in the paper represent surjective functions. In this case, greedy inductive compression of two examples of the same surjective function leads to the same hypothesis. This is demonstrated in the following cases, in which three examples are represented with distinct separators, *sep1* and *sep2*, which are presented to ReDuce within the same example.

Example set	$S \rightarrow \mathbf{c}, \mathbf{d}, \mathbf{c}, \mathbf{d}, sep1, \mathbf{c}, \mathbf{d}, \mathbf{c}, \mathbf{d}, \mathbf{c}, \mathbf{d}, sep2, \mathbf{c}, \mathbf{d}, \mathbf{c}, \mathbf{d}, \mathbf{c}, \mathbf{d}, \mathbf{c}, \mathbf{d}$
ReDuce	$S \rightarrow \mathbf{Star1}, sep1, \mathbf{Star1}, sep2, \mathbf{Star1}$

The ReDuce compressed sequence above show that the three example sequences are members of the same function,**Star1**. The approach appears to identify a *Normal Form* hypothesis (in this case **Star1**) by learning from a set of several examples. If this can be formally demonstrated it would avoid the potentially high costs associated with deductive testing of examples, since, by construction, each example generalises to the same hypothesis. Furthermore, negative examples and noisy data could be identified using the same mechanism as follows.

Example set	$S \rightarrow \mathbf{c}, \mathbf{d}, \mathbf{c}, \mathbf{d}, sep1, \mathbf{c}, \mathbf{d}, \mathbf{c}, \mathbf{d}, \mathbf{c}, \mathbf{e}, sep2, \mathbf{c}, \mathbf{d}, \mathbf{c}, \mathbf{d}, \mathbf{c}, \mathbf{d}, \mathbf{c}, \mathbf{d}$
ReDuce	$S \rightarrow \mathbf{Star1}, sep1, \mathbf{Star1}, \mathbf{c}, \mathbf{e}, sep2, \mathbf{Star1}$

In this case **Star1,c,e** represents either an error, a negative example or an exception to the general rule.

In further work the author aims to formally identify whether, when provided with multiple example sequences. Algorithm 1 always generates a unique hypothesis when provided with noise-free positive examples, allowing noisy examples to be identified.

Inputs and Outputs The experiments and implementation in this paper assume examples take the form of a sequence of primitives predicate calls, such as $revupc(\langle call1, pop, upcase, push, pop, upcase, push, pop, upcase, push, return1\rangle)$. This representation differs from DeepLog, which assumes examples take the form of Input/Output pairs such as *revupc([a,b,c],[’C’,’B’,’A’])*. DeepLog uses a two stage process involving 1) Meta-Compilation and 2) Meta-Interpretation. Meta-Compilation finds a set of minimal length sequences of primitives which transform the Input (eg *[a,b,c]*) to the Output (*[’C’,’B’,’A’]*). The minimal certificate is ⟨*call1,pop,upcase,push,pop,upcase,push, pop,upcase,push,return1*⟩. Meta-Interpretation then searches the hypothesis space to select a low complexity hypothesis which has low generality, in accordance with a MAP selection strategy. ReDuce presently only conducts a Meta-Interpretation stage.

In further work the author aims to extend ReDuce to include Deeplog’s Meta-compilation stage.

Large-scale testing In addition to further theoretical work on the identification of noise described above, further work is required to extend the benchmarks from the DeepLog paper. In particular, the author aims to test a broad range of tasks, including learning larger-scale programs as well as noisy real-world datasets. Error estimates will be introduced to better understand robustness and noise. Additionally the effects of scaling to large scale problems will be investigated.

Learning disjunctive concepts In further work the author aims to develop inductive compression with further underlying theory, implementations and experiments. We also intend to develop the representation and theory to address issues related to learning of arbitrary Prolog relations, such as *member* and *ancestor* from small numbers of examples.

Learning 2D grammars and images Lastly, the author hopes to explore inductive compression approaches for learning 2D grammars, as an approach to learning from images.

Acknowledgements

The author acknowledges support from both Nanjing University and the UK's EPSRC Human-Like Computing Network (EP/R022291/1), for which he acted as director. The author would also like to acknowledge helpful discussions on this topic with both Dr Lun Ai and Dr Wang-Zhou Dai. Lastly the author would like to thank his wife, Thirza, for her unfailing encouragement and willingness to always listen with care during the development of the work described.

References

1. Casel, K., Fernau, H., Gaspers, S., Gras, B., Schmid, M.: On the complexity of the smallest grammar problem over fixed alphabets. Theory of Computing Systems **65**, 344–409 (2021), https://doi.org/10.1007/s00224-020-10013-w
2. Gold, E.: Language identification in the limit. Information and Control **10**, 447–474 (1967)
3. McNeill, D.: Developmental psycholinguistics. In: Miller, G., McNeill, D. (eds.) Handbook of Social Psychology (1966)
4. Muggleton, S.: Duce, an oracle based approach to constructive induction. In: IJCAI-87. pp. 287–292. Kaufmann (1987), http://www.doc.ic.ac.uk/s̃hm/Papers/ijcai87.pdf
5. Muggleton, S.: Inductive Logic Programming. New Generation Computing **8**(4), 295–318 (1991), http://www.doc.ic.ac.uk/s̃hm/Papers/ilp.pdf
6. Muggleton, S.: Learning from positive data. In: Muggleton, S. (ed.) Proceedings of the Sixth International Workshop on Inductive Logic Programming (Workshop-96). pp. 358–376. LNAI 1314, Springer-Verlag, Berlin (1996), http://www.doc.ic.ac.uk/s̃hm/Papers/poslearn.pdf
7. Muggleton, S.: Hypothesising an algorithm from one example: the role of specificity. Philosophical Transaction of the Royal Society A **381:20220046** (2023), https://royalsocietypublishing.org/doi/10.1098/rsta.2022.0046

8. Muggleton, S., Lin, D., Pahlavi, N., Tamaddoni-Nezhad, A.: Meta-interpretive learning: application to grammatical inference. Machine Learning **94**, 25–49 (2014). https://doi.org/10.1007/s10994-013-5358-3, https://link.springer.com/article/10.1007/s10994-013-5358-3
9. Muggleton, S., Lin, D., Tamaddoni-Nezhad, A.: Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited. Machine Learning **100**(1), 49–73 (2015), https://link.springer.com/article/10.1007/s10994-014-5471-y
10. Muggleton, S., Raedt, L.D.: Inductive logic programming: Theory and methods. Journal of Logic Programming **19,20**, 629–679 (1994), http://www.doc.ic.ac.uk/s̃hm/Papers/lpj.pdf
11. Nevill-Manning, C., Witten, I.: Identifying hierarchical structure in sequences: A linear-time algorithm. Journal of Artificial Intelligence Research **7**, 67–82 (1997), https://arxiv.org/pdf/cs/9709102