Introducing WILLIAM: a system for inductive inference based on the theory of incremental compression

Arthur Franz* af@occam.com.ua
Michael Löffler* ml@occam.com.ua
Alexander Antonenko*† aa@occam.com.ua
Victoria Gogulya* vg@occam.com.ua
Dmytro Zaslavskyi*‡ dz@occam.com.ua

*Odessa Competence Center for Artificial intelligence and Machine learning (OCCAM) Odessa, Ukraine

†dept. of Mathematical Support of Computer Systems, Odessa National Mechnikov University, Odessa, Ukraine

‡Faculty of Mathematics, Physics and Information Technologies, Odessa National Mechnikov University, Odessa, Ukraine

***Abstract*—We introduce WILLIAM — a new system for data compression that is based on a formal mathematical theory of incremental compression. The theory promises to find short descriptions in an incremental and efficient way while still being applicable to a wide range of data. We have used abstract syntax trees of selected Python operators in order to define a representation language. We present some practical tests of the theory with encouraging results.**

***Index Terms*—incremental compression, data compression, Kolmogorov complexity, inductive inference**

I. Introduction

The ability to compress data is an important problem in both science and industry. Apart from saving memory space, data compression is tightly tied to inductive inference and artificial intelligence, since it requires intelligence in order to find regularities in data, which in turn leads to short representations (see [1], [2]).

Kolmogorov has proved that the concept of the shortest description is well defined and depends on the description language only up to an additive constant. The length of the shortest description is known as the (plain) Kolmogorov complexity of the data, which corresponds to optimal compression. In spite of substantial efforts in this area, progress is impeded by the fact that compression is possible only when a description, i.e. program, is found that captures regularities in the data. However, any language that is able to express all possible programs is Turing complete, such that the search within such a vast space becomes intractable.

In theory, there exist universal search methods, such as Levin Search, that are able to find short descriptions of data and require the execution of all lexicographically ordered programs until a solution is found. For the better or worse, Levin Search has the optimal order of computational complexity [3]. Nevertheless, the obvious slowness of this method, hidden in the big "O" notation, seems to be the price for its generality.

In order to alleviate that problem, we have developed a theory of incremental compression [4], that finds intermediate, increasingly shorter descriptions of data. Remarkably, it is guaranteed to find the shortest description (to some precision) by assembling it from short and mutually orthogonal features. In spite of the fact that those intermediate descriptions are incomputable, they can be approximated in practice, where the whole compression algorithm promises to be both general and efficient.

In this paper, we summarize the theoretical results expressed in the language of algorithmic information theory and also present a practical implementation of the proposed algorithm, WILLIAM: a Python-written system for data compression and inductive inference. Since this is work in progress, we include some modest results and discuss them in relation to the theory and future work.

II. Incremental compression

Consider strings made up of elements of the set $\mathcal{B} = \{0, 1\}$ with ϵ denoting the empty string. $\mathcal{B}^*$ denotes the set of finite strings over $\mathcal{B}$. Denote the length of a string x by $l(x)$. Since there is a one-to-one map $\mathcal{B} \leftrightarrow \mathcal{N}$ of finite strings onto natural numbers, strings and natural numbers are used interchangeably.

The universal, prefix Turing machine U is defined by

$$U\left(\langle y, \langle i, p\rangle\rangle\right) = T_i\left(\langle y, p\rangle\right), \tag{1}$$

where T_i is i-th machine in some enumeration of all prefix Turing machines and $\langle ., .\rangle$ is some one-to-one map $\mathcal{N} \times \mathcal{N} \leftrightarrow \mathcal{N}$. This means that $f = \langle i, p\rangle$ describes some program on a prefix Turing machine and consists of the number i of the prefix Turing machine and its input parameter p and y is some additional parameter. We will use the shortcut $U\left(\langle x, f\rangle\right) \equiv f(x)$. The conditional (prefix Kolmogorov) complexity is given by

$$K(x|y) := \min_s \left\{l(s) :\ U\left(\langle y, s\rangle\right) = s(y) = x\right\} \tag{2}$$

This means that $K(x|y)$ is equal to the shortest description of string x given string y on a universal, prefix Turing machine. The unconditional (prefix Kolmogorov) complexity is defined by $K(x) \equiv K(x|\epsilon)$. Up to this point, we have followed the standard definitions as given in e.g. [2].

Our theory of incremental compression has been presented in [4], which we summarize here.

Definition 1: Let f and x be finite strings and $D_f(x)$ the set of **descriptive maps** of x given f:

$$D_f(x) = \{f' : f(f'(x)) = x, l(f) + l(f'(x)) < l(x)\} \quad (3)$$

If $D_f(x) \neq \emptyset$ then f is called a **feature** of x. The strings $p \equiv f'(x)$ are called **parameters** of the feature f. f^* is called **shortest feature** of x if it is one of the strings fulfilling

$$l(f^*) = \min\{l(f) : D_f(x) \neq \varnothing\} \quad (4)$$

and f'^* is called **shortest descriptive map** of x given f^* if

$$l(f'^*) = \min\{l(g) : g \in D_{f^*}(x)\} \quad (5)$$

Lemma 1:

Let f^* and f'^* be the shortest feature and shortest descriptive map of a finite string x, respectively. Further, let $p \equiv f'^*(x)$. Then

1) $l(f^*) = K(x|p)$ and
2) $l(f'^*) = K(p|x)$.

Theorem 1 (Feature imcompressibility):

The shortest feature f^* of a finite string x is incompressible:

$$l(f^*) - O(1) \leq K(f^*) \leq l(f^*) + O(\log(l(f^*))). \quad (6)$$

Theorem 2 (Independence of features and parameters):

Let f^* and f'^* be the shortest feature and descriptive map of a finite string x, respectively. Further, let $p \equiv f'^*(x)$. Then,

$$K(f^*|p) \approx K(f^*), \quad (7)$$

$$K(p|f^*) \approx K(p), \quad (8)$$

$$K(f^*, p) \approx K(f^*) + K(p) \quad (9)$$

where the "$\approx$" sign denotes equality up to logarithmic terms in complexity.

We conclude that features and parameters do not share information about each other, therefore the description of the (f^*, p)-pair breaks down into the simpler task of describing f^* and p separately. Since Theorem 1 implies the incompressibility of f^* and $U(\langle p, f^* \rangle) = x$, the task of compressing x is reduced to the mere compression of p.

Let us describe the compression scheme of the binary string x. Denote $p_0 \equiv x$, and start an iterative process of compression: let f^*_{i+1} be a shortest feature of p_i, f'^*_{i+1} is a shortest corresponding descriptive map and $p_{i+1} = f'^*_{i+1}(p_i)$. We will continue this process until some p_s is not compressible (for example, $p_s = \epsilon$) and obtain $x = f^*_1(f^*_2(\cdots f^*_s(p_s)))$. One of the main theoretical results shows that this representation approximates the Kolmogorov complexity up to logarithmic terms, i.e. achieves near optimal compression.

III. Description of the algorithm

In the following we will describe the current state of WILLIAM, which is work in progress and will change considerably in future. The main goal of this project is to implement our theory of incremental compression in practice and to construct intelligent agents using the inductive reasoning capabilities that follow from the ability to compress data. For example, it has been shown formally that combining optimal compression (in the form of Solomonoff induction [5], [6]) with reinforcement learning leads to maximally intelligent agents [1].

A. The alphabet

As a language we have used trees of Python operators, which can be converted to abstract syntax trees, compiled and executed by Python itself. At the core, we use an alphabet of currently 36 Python operators, such as `range`, `add`, `mult`, `join`, `map`, `equal`, `and`, `or`, `ifelse` etc. Each operator can be called and knows its arity and type specifications, i.e. the types of variables that can be it's input and output. The currently allowed types are integer, float, string, bool and callable functions, which can be put into lists and tuples. Each operator knows whether and how it can be inverted. For example, `range(4) = [0,1,2,3]`, thus given the list as *induction target*, the input `4` can be inferred. Often, an operator can only be inverted if *conditioned* on some of its inputs. For example, `add(3,4) = 7` can be inverted, if the target `7` is given and the first or second input is conditioned on (i.e. it can be solved for the other input). Sometimes inversion leads to several solutions. For example, if `concat(a,b) = [0,1,2]`, then possible solutions are `a = [], b = [0,1,2]`; `a = [0], b = [1,2]`; `a = [0,1], b = [2]` or `a = [0,1,2], b = []`. Therefore, the inversion of operators is implemented as a Python generator, which yields all possible results. The resulting language defined by that alphabet is completely functional (not declarative).

B. Composite operators

The alphabet can be dynamically extended by defining trees made up of them, so that a tree can act as a single operator. Such new operators are called *composite operators*. They can be executed and also inverted depending on the invertibility and conditions of the participating operators at the nodes. In order to invert a composite, it has to be computed, which if its inputs have to be given, i.e. are conditioned and which can be computed. For example, the composite function $y = x_1 - x_2 + x_3 * x_4$ can be inverted given 3 out of 4 variables, which lead to the inverse functions $F_1 = (y - x_1 + x_2)/x_4$, $F_2 = (y - x_1 + x_2)/x_3$, $F_3 = y - x_1 - x_3 * x_4$ and $F_4 = y + x_2 - x_3 * x_4$. A special activation propagation algorithm has been deployed in order to implement such composite inversions. Composites can then be reused as single operators at the nodes of even larger trees – *hypertrees*.

C. Tree search

Given the alphabet with its respective arity and type specifications an exhaustive search for trees made up of those operators can be performed with the constraint that the output of an operator is fed into an input of another operator with a compatible type specification. Two versions of tree search have been implemented. The first is a depth-first search of all perfect trees at given depth. The second version is a description length sorted (=biased) tree search where trees of increasing description length are found. This

is important since we want to find simple trees first. For example, a degenerate, non-branching depth-4-tree of four unary operators may be simpler than a general depth-3-tree.

D. Search for parameters

After some tree has been constructed, it is used as a single composite operator and corresponds to a feature, as defined in our theory. WILLIAM tries to find inputs (=parameters) to the composite tree (=feature), such that the output matches the target. There are three ways of finding parameters that are currently implemented.

The first and the least efficient, tries to cycle through all combinations of parameters. For example, if the composite requires 5 integer parameters then all integer combinations up to a certain maximal size are attempted.

The second way uses the reasoning of the incremental compression theory: it uses the composite operator (feature f) and the target x in order to *compute* the parameters p. In general, some parameters have to be conditioned on, while the others can be inferred, which corresponds to solving an equation for unknown variables. WILLIAM takes those parameters that have to be known and computes all combinations like in the first version and infers all the other parameters by inverting the composite. Essentially, the descriptive map f' is given by the conditioned parameters p_c and the composite tree itself (for example, inversion of $\texttt{add}(p_c, p_i) = 7$ is possible, if p_c is known).

The third way is to use so-called biased permutation, where the generation of parameters is sorted by their description length.

E. Description length

The description length of various data sets is computed in the following way. Integers n are encoded with the Elias delta code [7], whose length is

$$l(n) = \lfloor \log_2(n) \rfloor + 2 \lfloor \log_2 (\lfloor \log_2(n) \rfloor + 1) \rfloor + 1 \quad (10)$$

Floats are approximated by fractions up to a given precision by minimizing the sum of the description lengths of nominator and denominator, which are integers. Chars are described by the Elias code of their ASCII number. The description length of iterable structures such as strings, lists and tuples consisted of basic elements is simply the sum of the lengths of each element plus description length of the length of the iterable structure.

Beyond data sets, elementary operators and the trees of them that define composite operators have to be described. The information needed to specify an elementary operator is simply $\lceil \log_2(N) \rceil$ where N is the length of the alphabet, currently $N = 36$. Since each operator knows the number if its inputs/children, a tree made up of operators can be described by assigning a number $0, \ldots, N-1$ to each operator and writing those numbers in sequence, assuming a depth-first enumeration order.

F. Inductor and incremental compression

Currently, WILLIAM can function in two modes. In the universal search mode, WILLIAM acts as simple inductor that find descriptions of a target string x in the form of an operator tree f and its parameters p, while trying to minimize the total description length $l(f) + l(p)$. In the incremental mode, WILLIAM tries to find trees f with short description lengths and compute parameters by inverting the tree. Here, only $l(f)$ is minimized while p is allowed to be long, merely bounded by the compression condition $l(f) + l(p) < l(x)$. The parameter list then becomes a new target.

More precisely, let x be an initial target, which is represented in the form of $x = f_1(p_1)$ where f_1 is an operator tree and p_1 is a list of parameters fulfilling $l(f_1) + l(p_1) < l(x)$. This procedure is repeated: $p_1 = f_2(p_2)$, $p_2 = f_3(p_3)$ etc. in accordance with the idea of incremental compression to use short features. In this way, we obtain a list of trees that can be executed subsequently, and form a composition of functions $f_1(f_2(\cdots f_s(p_s)))$ each fulfilling the compression condition $l(f_{i+1}) + l(p_{i+1}) < l(p_i)$. We call such lists/rows of trees *alleys*.

IV. Experiments

In the following, we present some test cases of what has been achieved.

A. Examples of induced functions

All trees up to depth 2 were searched through. We choose some target, then run the inductor in the universal search mode and give some examples of induced functions, the number of attempts to find this representation and compression ratio (in percents of target description length), see *Table I*. The compression ratio is required to be greater than zero, since we enforce the compression condition. w/hints means that we have used hints by limiting the set of basic functions in the inductor. In the condition "without inversion" all parameter combinations up to a certain threshold have been searched through exhaustively. In the "with inversion" condition, some of the parameters can be inferred by computing them from the target and from the conditioned inputs.

It is quite evident that guessing both the function tree and appropriate inputs to the tree is quite computationally expensive. However, when inputs can be computed by inversion, the search is much faster and also can solve problems that could not be solved before. This reflects the $p = f'(x)$ operation in the theory.

B. Example of an induced alley

Consider the target x from *Table II*. It is complex enough, such that an exhaustive search for a representation quickly becomes intractable. However, in the incremental compression mode, WILLIAM can still find a solution. It has first found a function $f_1(a, b, c, d)$ (in the form of an operator tree) and its parameter list $p_1 = [a, b, c, d]$. Both f_1 and p_1 are given in *Table II* and together they form the representation

$$x = \texttt{insert}\left(\texttt{range}\,(0, 6), 11, [12, 13, \ldots, 22, 8, \ldots, 8]\right),$$

which is shorter than the initial target x. In the current version of WILLIAM, the new target is a concatenated list of parameters (denoted by $\mathrm{c}(p_1)$), so at step two the new target is set to

$$\mathrm{c}(p_1) = [0, 6, 11, 12, 13, \ldots, 22, 8, \ldots, 8].$$

TABLE I
EXAMPLES OF INDUCED FUNCTIONS

Target	Some induced function	Attempts without inversion	Attempts with inversion	Compression
'111111'	str(111111), *(6,'1')	not found, 1290	4, 8	53%, 57%
[1,2,4,8,16,32,64,128,256]	power(2, range(9))	669071	196	70%
'aaaaazzzzz'	5 * 'a' + 5 * 'z'	not found	2136	53%
[0.0,1.0,...,99.0]	map(float,range(100))	866, w/hints	3, w/hints	97%
[33,35,37,39,15,14,...,6]	range(33,41,2)+range(15,5,-1)	not found	3021	49%

TABLE II
EXAMPLE OF AN ALLEY

Denotation	Function	List of parameters
x		[11,11,11,11,11,11,12,13,14,15,16,17,18,19,20,21,22,8,8,8,8,8]
$f_1(a,b,c,d), p_1$	insert(range(a,b),c,d)	[0,6,11,[12,13,14,15,16,17,18,19,20,21,22,8,8,8,8,8]]
$f_2(a,b,c,d), p_2$	insert(range(a,b),c,d)	[14,19,8,[0,6,11,12,13,14,15,16,17,18,19,20,21,22]]
$f_3(a,b,c,d), p_3$	insert(range(a),b,range(c,d))	[5,[14,19,8,0,6],11,23]

After running the inductor on the new target we obtain a new feature $f_2(a,b,c,d)$ and its parameter list p_2 such that $\mathrm{c}(p_1) = f_2(p_2)$. Using $\mathrm{c}(p_2)$ as the new target we obtain $f_3(a,b,c,d)$ and p_3. This process can be continued but the inductor did not find a shorter representation of $\mathrm{c}(p_3)$ in this example. Overall, the final description of the target x contains features (functions) f_1, f_2, f_3, a parameter p_3 and some information that allows to obtain initial versions of p_i from the concatenated forms $\mathrm{c}(p_i)$, by saving the indices of each parameter in the concatenated list $\mathrm{c}(p_i)$.

V. DISCUSSION

Previously, we have developed a theory of incremental compression that promises the existence of an efficient and general data compression algorithm. In this publication, we have sketched our new system WILLIAM that aspires to realize that algorithm in practice. WILLIAM is much more complex than described here, but a full description of its functionality is beyond the scope of this paper. Still, it is very much work in progress, so that we have been able to demonstrate some modest capabilities for inducing short descriptions on some examples without yet being able to run a systematic comparison with industrial standards for compression algorithms. Also, a detailed match to the theory in order to test the derived theorems would be useful as well as the further development of a computable version of the theory. Nevertheless, the inductive capabilities using alleys have shown the emergence of fairly deep trees, which would be practically impossible to find using exhaustive search. This is an encouraging confirmation of the theory's efficient compression abilities.

REFERENCES

[1] M. Hutter, *Universal Artificial Intelligence: Sequential Decisions based on Algorithmic Probability*. Berlin: Springer, 2005. 300 pages, http://www.hutter1.net/ai/uaibook.htm.

[2] M. Li and P. M. Vitányi, *An introduction to Kolmogorov complexity and its applications*. Springer, 2009.

[3] L. A. Levin, "Universal sequential search problems," *Problemy Peredachi Informatsii*, vol. 9, no. 3, pp. 115–116, 1973.

[4] A. Franz, "Some theorems on incremental compression," in *International Conference on Artificial General Intelligence*, pp. 74–83, Springer, 2016.

[5] R. J. Solomonoff, "A formal theory of inductive inference. Part I," *Information and control*, vol. 7, no. 1, pp. 1–22, 1964.

[6] R. J. Solomonoff, "A formal theory of inductive inference. Part II," *Information and control*, vol. 7, no. 2, pp. 224–254, 1964.

[7] P. Elias, "Universal codeword sets and representations of the integers," *IEEE transactions on information theory*, vol. 21, no. 2, pp. 194–203, 1975.