WILLIAM: A Monolithic Approach to AGI

Arthur Franz(✉), Victoria Gogulya, and Michael Löffler

Odessa Competence Center for Artificial intelligence and Machine learning (OCCAM), Odesa, Ukraine
{af,vg,ml}@occam.com.ua

Abstract. We present WILLIAM – an inductive programming system based on the theory of incremental compression. It builds representations by incrementally stacking autoencoders made up of trees of general Python functions, thereby stepwise compressing data. It is able to solve a diverse set of tasks including the compression and prediction of simple sequences, recognition of geometric shapes, write code based on test cases, self-improve by solving some of its own problems and play tic-tac-toe when attached to AIXI and without being specifically programmed for it.

Keywords: Inductive programming · Incremental compression · Algorithmic complexity · Universal induction · AIXI · Seed AI · Recursive self-improvement

1 Introduction

It is well-known to any trained machine learning practitioner that the choice of a learning algorithm severely depends on the class of tasks to be solved. Cynical voices have even claimed that the really intelligent part of the final performance actually resides within the human developer choosing, twisting and tweaking the algorithm, rather than inside the algorithm itself.[1] Somehow, a trained human is able to recognize the narrowness of the algorithm and come up with simple data samples which the algorithm will fail to represent.

In the context of AGI we would like to ask, how can we build an algorithm that can derive such wide and diverse representations that even trained humans will not recognize their limits? Conversely, humans seem to be limited by the complexity of the represented data, since the ability to recognize structure in large and complex data sets is what we need machine learning for in the first place. The strength of those algorithms appears to entail their weakness: the inability to represent a wide range of simple data. In order to make progress toward AGI, we suggest to fill the "cup of complexity" for the bottom up, as exemplified in Fig. 1.1 in [3]. Instead of building algorithms for complex but narrow data, we suggest heading for simple but general ones.

[1] Christoph von der Malsburg, personal communication.

This is exactly what we have attempted in the present paper: we have built WILLIAM – an algorithm that can handle a diverse set of simple data with the goal of turning up the complexity in future. The importance of this approach is also backed by the idea of a seed AI, recursively self-improving itself. After all, in order to f-improve, an algorithm has to be able to solve problems occurring during its own implementation. Even more so, the fraction of issues solved by the algorithm itself rather than by the programmers should increase continuously during the development of a real AGI system. Instead, developers usually build various specialized algorithms, i.e. heuristics, in order to deal with various difficulties during development, leading to narrow and brittle solutions. We have avoided to use heuristics as much as possible and demonstrate in Sect. 3.5 a simple self-improving ability of WILLIAM.

As opposed to cognitive architectures which consist of many narrow algorithms trying to patch up each others representational holes in the crusade against the curse of dimensionality, WILLIAM constitutes a monolithic approach, i.e. it is essentially a single algorithm that helps itself and stays out of high-dimensional regions of the task space by systematically giving preference to simple data. Such a strong Occam bias is also what is demanded by formal descriptions of optimal prediction and intelligence [6,14]: the prior probability of a representation q of length $l(q)$ being relevant to the current task should be given a weight of $2^{-l(q)}$!.

These considerations impose several constraints on any monolithic algorithm. In order to be wide enough it should employ search of general programs expressed in a Turing complete language. Those programs have to be as simple as possible, hence achieve high data compression ratios. WILLIAM achieves this by searching in the space of syntax trees made up of Python operators and building stacked autoencoders. By doing so, it exploits the efficiency of optimal compression guaranteed by the theory of incremental compression [4]. In order to enable actual problem solving, we have attached these compression abilities to the expectimax search of actions leading to high expected rewards, as demanded by an optimally intelligent AIXI agent [6]. In Sect. 3.6 we describe how the resulting agent was able to play tic-tac-toe without being explicitly programmed for it.

2 Description of the Algorithm

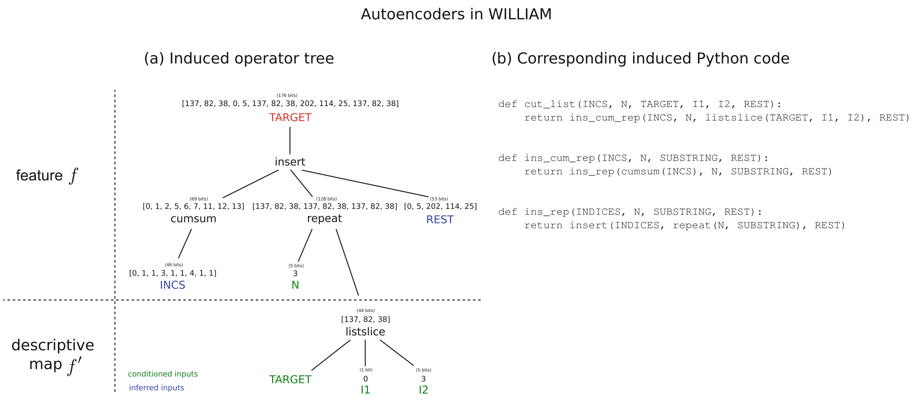
In the following we will describe the current state of WILLIAM built in our OCCAM laboratory during the last 1.5 years. It is work in progress and consists currently of about 9600 lines of Python code, including 876 test cases. Its main task is, given a data string, to induce a representation with a short description length. As a language we have used trees of currently 41 Python operators, which can be converted to abstract syntax trees, compiled and executed by Python itself. We have described the basics of the algorithm already in [5] and will focus on the novel parts in this paper.

2.1 Description Length

The description length of various data sets is computed in the following way. Positive integers n are encoded with the Elias delta code [1], whose length is

$$l(n) = \lfloor \log_2(n) \rfloor + 2 \lfloor \log_2 (\lfloor \log_2(n) \rfloor + 1) \rfloor + 1 \tag{1}$$

Arbitrary integers are assigned the code length $l(2|n|+1)$, due to the extra bit for the sign. 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 cost 8 bits due to 256 elements of the ASCII table. 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. For example, the description length of `[-21,7,-4]` is $l(2{\cdot}21{+}1){+}l(2{\cdot}7{+}1){+}l(2{\cdot}4{+}1){+}l(3) = 10{+}8{+}8{+}4 = 30$ bits.



```
def cut_list(INCS, N, TARGET, I1, I2, REST):
    return ins_cum_rep(INCS, N, listslice(TARGET, I1, I2), REST)


def ins_cum_rep(INCS, N, SUBSTRING, REST):
    return ins_rep(cumsum(INCS), N, SUBSTRING, REST)


def ins_rep(INDICES, N, SUBSTRING, REST):
    return insert(INDICES, repeat(N, SUBSTRING), REST)
```

Fig. 1. (a) The target list contains a subsequence `[137,82,38]` repeating three times. The autoencoder tree squeezes the target through an information bottleneck. (b) The tree corresponds to actual Python code that can be compiled and executed.

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 $\lceil \log_2(N) \rceil = 6$ bits where N is the length of the alphabet, currently $N = 41$. 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. Thus, the description length of a tree is simply 6 times the number of operators.

2.2 Representing Autoencoders

The idea of incremental compression rests on the construction of stacked autoencoders. Figure 1(a) exemplifies one such autoencoder. The whole autoencoder is a single tree where the target has to be the output of the tree and is allowed to be an input to any of the leaf nodes of appropriate type. When a tree with the property is found, the system tries to cut through it at the location of smallest description length, i.e. it tries to find the information bottleneck whose description length (called *waistline*) is shorter than the target. In Fig. 1(a) `listslice` and its inputs `I1` and `I2` slice the first three elements from the target list, functioning in the role of a descriptive map f'. The information bottleneck is given by the dashed line at which point the intermediate, residual description[2] is `[INCS,N,[137,82,38],REST]`. The description length of the waistline consists of $46 + 3 + 44 + 53 = 146$ bits for the residual description and 18 bits for the feature tree (6 bits per operator)). This is less than the description length of the target (176 bits). Ergo, some compression has been achieved. The feature tree f decompresses those variables back onto the target by inserting the repeating subsequence at the indicated indices. The conditioned inputs are those to be given (i.e. guessed), the inferred inputs can be computed by inverting the tree. We see that without the descriptive map f', i.e. `listslice`, the list `[137,82,38]` would have to be guessed in order to compress the target, which is highly improbable.

The search for such autoencoders is performed by enumerating all trees of operators from the set of primitives, such that input and output types match. The enumeration was sorted by a rating consisting of the description length of the tree and conditioned inputs plus the logarithm of the number of attempts to invert that tree, i.e. to find some inferred inputs. When some tree can not be inverted for a long time, its rating drops and the search focuses on trees with the smallest rating. Effectively, this search amounts to exhaustive search for a pair (f, f') sorted by the sum of their lengths $l(f) + l(f')$ if we consider the conditioned inputs as belonging to f'.

After an autoencoder is found, f' is removed and the residual description is joined to a single list which serves as the target for the next autoencoders to be stacked on the first one (see Sect. 3.2 for more details). This incremental process continues until either no compression can be found or some maximal search time is reached.

2.3 Prediction

Given a function and its inputs, predictions are generated by manipulating the inputs in certain ways. In general, as demanded by Solomonoff induction, every possible continuation of a string has to be considered and separately compressed. Remember the Solomonoff prior of a string x:

$$M(x) = \sum_{q:U(q)=x} 2^{-l(q)} \tag{2}$$

where $x, q \in \mathcal{B}^*$ are finite strings defined on a finite alphabet $\mathcal{B}$, U is a universal Turing machine that executes program q and prints x and $l(q)$ is the length of

[2] What we have called "parameter" in [4] is now called "residual description".

program q. Given already seen history $x_{<k} \equiv x_1 \cdots x_{k-1}$ the prediction of the next h bits is computed by considering all $|\mathcal{B}|^h$ strings of length h and computing the conditional distribution $M\left(x_{k:k+h-1}|x_{1:k}\right) = M\left(x_{<k+h-1}\right)/M\left(x_{<k}\right)$, each of which requires to find a set of short functions q that are able to compute $x_{<k+h-1}$.

This is clearly intractable in practice. However, if we consider the set of short programs for $x_{<k+h-1}$ in which the continuation $x_{k:k+h-1}$ is chosen freely, it can be shown that the continuation can not contain much additional information compared to $x_{<k}$[3]. Therefore, that set of programs is likely to be among the set of shortest programs already found for $x_{<k}$. In this sense, although it is not a general solution, it appears reasonable to look for small modifications of a short function of $x_{<k}$ in order to predict its continuation.

In the present implementation we modify those leaves of the tree that are related to the length of the output sequence, trying to extend its length. This seems like a heuristic, however, from the theoretic perspective it turns out that given a feature f of x, to arrive at some extended string xy, it takes a program of constant size, $K(g|f) = O(1)$, to arrive at some feature g of xy (yet unpublished result). In that sense, we are well-set for the computation of predictions in a theoretically grounded way.

In this paper, we use the shortest found functions for computing predictions, but nothing prevents us from using several short functions for the approximation of the Solomonoff prior and the Bayesian posterior for prediction purposes. For prediction examples see Table 1 below.

Table 1. Examples of induced functions. The indicated compression ratio has been reached after the indicated number of program execution attempts.

Target	Induced program	Attempts	Compression	Predictions
`[0,1,2,...,99]`	`urange(100)`	1	98%	`100,101,102,103,...`
`[0,9,9,1,9,9,2,9,9, ...,6,9,9]`	`insertel(trange(0,21,3), urange(7),21,9)`	1383	62%	`7` or `9,9,...`
`[9,8,9,9,8,9,8,8,9,9,9]`	`cumsum([9,-1,1,0,-1,1, -1,0,1,0,0])`	2	51%	`9` or `10` or `11` or `12`...
`[100,200,300,400,500, 100,200,...,500]`	`repeat(100, trange(100,600,100))`	46	99%	`100,200,300,400,500, 100,200,...`
`[7,1,8,7,1,8,7,..., 1,8,3,3,...,3,3]`	`conc_rep(20, [7,1,8], repeat(15, [3]))`	317	85%	`3,3,3,...`
`[33,35,...,147,149,150, 149,148,...,6,5]`	`conc_tr(33, 151, 2, trange(150, 4, -1))`	384	96%	`4,3,2,...`
`'ABCDEFGHIJKLMN'`	`str2idx(trange(65,79,1))`	23	60%	‘`OPQRS...`’
`'aaaaammmmmzzzzz'`	`str2idx(cumsum([[97,0,0,0, 0,12,0,0,0,0,13,0,0,0,0]]))`	8	49%	‘`z`’ or ‘`{`’ or ‘`|`’...

[3] The proof would be beyond the scope of the present paper and will be published soon.

3 Results

3.1 Examples of Induced Programs

In the following, we present some test cases of what has been achieved so far. The algorithm keeps searching for matching programs, until either a configurable amount of valid programs has been found, or all functions up to a certain description length have been attempted. The *target* column in Table 1 describes the data, for which the algorithm had to find a program, consisting of a function and input data for that found function, so that calling the function with that input data reproduces the target. In the last four examples, we also demonstrate the reuse of already found functions, which were not present in the original set of primitives, but were previously learned and (manually) stored.

In order to compute predictions, each function (also called *composite operator*) knows which leaves to update in order to extend the output. For example, in the ABCD.. example, 79 is updated to 80 etc. since the `trange` operator knows that its second input has to be incremented in order to increment its output. Every operator knows how to be incremented which is propagated to the whole composite. Integers are incremented by 1, while lists are extended by `[0]` or `[1]` or `[2]` etc. which in the `cumsum` example in the third line, leads to the predictions 9, 10, 11 etc.

Table 2. Example of an incrementally induced composition of functions (also called *alleys* in [5])

Denotation	Feature	Residual description	Compression
x		`[0,9,9,1,9,9,2,9,9,3,9,9,4,9,9,5,9,9,6,9,9]`	0%
$f_1(a,b,c), r_1$	`insertel(a,b,c)`	`[[0,3,6,9,12,15,18], [0,1,2,3,4,5,6],21,9]`	28%
$f_2(a), r_2$	`cumsum(a)`	`[[0,3,3,3,3,3,3,-18,1,1,1,1,1,1,15,-12]]`	40%
$f_3(a), r_3$	`cumsum(a)`	`[[0,3,0,0,0,0,0,-21,19,0,0,0,0,0,14,-27]]`	52%

3.2 Example of Incrementally Induced Composition of Functions

Consider the target x from Table 2. WILLIAM has first found a function $f_1(a,b,c)$ and its residual description $r_1 = [a]$. The feature f_1 and the residual description r_1 form the program

$$x = \texttt{insertel}([0,3,6,9,12,15,18],[0,1,2,3,4,5,6],21,9), \tag{3}$$

which inserts the numbers 0 to 6 at the indicated indices and fills the rest with 9's, which is shorter than the initial target x by 28%. In the current version of WILLIAM, the new target is obtained by joining the residual into a single list (denoted by $\mathrm{c}(r_1)$), so at step two the new target is set to

$$\mathrm{c}(r_1) = [0,3,6,9,12,15,18,0,1,2,3,4,5,6,21,9].$$

This concatenation is done in order to account for possible mutual information between leaves. For example, the two lists in Eq. (3) both increment the previous number by a constant and therefore there is mutual information between those lists. Therefore, the application of a single `cumsum` function to the concatenated list $c(r_1)$ compresses both lists.

After running the inductor on the new target $c(r_1)$ we obtain a new feature $f_2\,(a)$ and its residual r_2 such that $\mathrm{c}(r_1) = f_2\,(r_2)$. Using $\mathrm{c}(r_2)$ as the new target we obtain $f_3\,(a)$ and p_3. This process is continued until the inductor can not find a shorter description, which is $\mathrm{c}(p_3)$ in this example. Overall, the final description of the target x contains features (functions) f_1, f_2, f_3, a residual r_3 and some information that allows to obtain initial versions of r_i from the concatenated forms $\mathrm{c}(r_i)$, by saving the indices of each input in the concatenated list $\mathrm{c}(r_i)$.

This compression required merely 45 attempts while the non-incremental compression of the same required 1383 attempts (see Table 1), demonstrating how incremental compression can be faster than non-incremental compression. However, the compression ratio is somewhat lower. The reason is the concatenation of r_i to $c(r_i)$, which deletes the information about where the input to one leaf ends and to the next leaf starts. We could do without this concatenation and incrementally compress the values at each leaf independently, but this would ignore possible mutual information between leaves. Consider another example in Table 3, where this effect is even stronger.

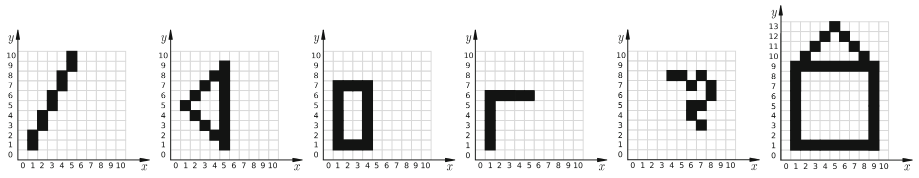
Table 3. Another example of incremental compression. The depth of the incrementally found tree is 7, which would be intractable in the non-incremental setting.

Denotation	Feature	Residual description	Compr.
x		`[7,1,8,7,1,8,...,7,1,8,3,3,...,3]`	0%
$f_1\,(a,b)\,,r_1$	`table(a,b)`	`[[1,3,7,8], [2,0,3,2,0,3,...,2,0,3,1,1,...,1]]`	36%
$f_2\,(a,b,c,d,e)\,,r_2$	`insertel(trange(a,b,c),d,e)`	`[1,64,1, [3,7,8,2,0,3,2,0,3,...,2,0,3], 79, 1]`	38%
$f_3\,(a,b,c,d,e)\,,r_3$	`insert(trange(a,b,c),d,e)`	`[6,66,3,2, [1,64,1,3,7,8,0,3,0,3,...,0,3,79,1]]`	49%
$f_4\,(a,b,c,d,e)\,,r_4$	`insert(trange(a,b,c), d,cumsum(e))`	`[10,50,2,0, [6,60,-63,-1,-1,63,-63,2,4,1,-5,0,0,...,0,76,-78]]`	57%

This very target has been compressed by 85% in Table 1, while incremental compression has only achieved 57%. We see that due to the just discussed concatenation of residuals, the next feature has to make efforts to cut out the compressible parts of the residual using the `insert` and `trange` operators, wasting description length this way. We don't have a satisfactory solution to this problem at this point. Nevertheless, this example demonstrates that deep trees can be found incrementally.

3.3 Perception: Recognition of Geometric Figures and Line Drawings

More examples are seen in Fig. 2 where a list of coordinate pairs coded for various geometric figures. Those figures could be compressed successfully by



Double-pointed diagonal line:
[(1, 1), (1, 2), (2, 3), (2, 4), (3, 5), (3, 6), (4, 7), (4, 8), (5, 9), (5, 10)]

Triangle:
[(5, 1), (4, 2), (3, 3), (2, 4), (1, 5), (2, 6), (3, 7), (4, 8), (5, 9), (5, 8), (5, 7), (5, 6), (5, 5), (5, 4), (5, 3), (5, 2)]

Rectangle:
[(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7), (2, 7), (3, 7), (4, 7), (4, 6), (4, 5), (4, 4), (4, 3), (4, 2), (4, 1), (3, 1), (2, 1)]

Corner:
[(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (2, 6), (3, 6), (4, 6), (5, 6)]

Path:
[(4, 8), (5, 8), (6, 7), (7, 8), (8, 7), (8, 6), (7, 5), (6, 5), (6, 4), (7, 3)]

House:
[(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7), (1, 8), (1, 9), (2, 9), (3, 9), (4, 9), (5, 9), (6, 9), (7, 9), (8, 9), (9, 9), (9, 8), (9, 7), (9, 6), (9, 5), (9, 4), (9, 3), (9, 2), (9, 1), (8, 1), (7, 1), (6, 1), (5, 1), (4, 1), (3, 1), (2, 1), (2, 10),(3, 11), (4, 12), (5, 13), (6, 12), (7, 11), (8, 10)]

Fig. 2. Incrementally compressed simple geometric figures, paths and line drawings

functions like `zip(cumsum(cumsum(y))`. They also constitute examples of incremental compression in the sense that the list of coordinates were transformed to two lists of $x-$ and $y-$values, respectively, by an autoencoder using the `zip` operator. Remarkably, the residual description does not correspond to any usual way, we would represent a, say, rectangle. Usually, we would encode the coordinates of one corner `(x0,y0)`, the length `dx` and the width `dy`. Instead, consider a larger version of the rectangle in Fig. 2, having a width `dx=100` and height `dy=200` with one corner defined by `x1=850` and `y1=370`. The residual after incremental compression turns out to be

```
[[0,1,dx+1,n+1,dx+n,2*n,2*n+1,dx+2*n+1,3*n,dx+3*n],
[y0,-y0,1,-1,-1,x0-y0+1,y0-x0+1,-1,-1,1],4*n,0],
```

where `n=dx+dy`. We see that the residual does consist of the basic parameters of a rectangle and some small numbers, albeit not fully compressed. Moreover, we see that the residual of a square is a special case of the residual of a rectangle, where `dx=dy`. The reason why compression has stopped before is the compression condition: the residual has become quite short such as the description length of a new feature plus its residual is too long. The fundamental reason is, that in order to compress optimally, we should not consider a single rectangle, but a set of rectangles compressed by the same function (see discussion). Overall, these examples demonstrate that WILLIAM can compress simple geometric figures, paths and line drawings.

3.4 Combining Our Induction Algorithm with AIXI

Given some induction capabilities of WILLIAM, we have implemented a plain version of the AIXI agent, which chooses its actions according to the formula [6]

$$a_k := \arg\max_{a_k} \sum_{o_k r_k} \cdots \max_{a_m} \sum_{o_m r_m} [r_k + \cdots + r_m] \cdot \sum_{q:U(q,a_1 \ldots a_m)=o_1 r_1 \cdots o_m r_m} 2^{-l(q)} \tag{4}$$

where a_i, o_i and r_i are actions, observations and rewards at time step i, respectively. In a nutshell, AIXI has seen the observation and reward sequence $o_1 r_1 \cdots o_{k-1} r_{k-1}$, performed actions $a_1 \cdots a_{k-1}$ and considers all permutations of futures $o_k r_k \cdots o_m r_m$ and actions $a_k \cdots a_m$. For each such permutation it looks for all programs, executed on a universal Turing machine U that can compute the observation-reward sequence given the actions. The shortest such programs receive the highest weight $2^{-l(q)}$, which implies that the most predictive programs are predominantly considered. Additionally, if the reward sum for such a highly probable future is high as well, the action maximizing this expected reward is taken.

WILLIAM uses its inductive capabilities in order to find a list of programs q, which are operator trees, and computes their description length $l(q)$. The expectimax tree is implemented as is, without any modifications. The next two subsections present examples of intelligent tasks solved by WILLIAM.

3.5 Recursive Self-improvement: WILLIAM Helps Itself to Do Gradient Descent

One of the most thorny induction problems is the search for good sampling algorithms. The central problem is to find inputs to a given function, such that its value is maximized. For example, probabilistic graphical models face the problem of finding areas of high probability in a high-dimensional space [9]. A long list of sampling procedures have been researched, however, a general solution is not in sight. Any procedure is good for some problems and bad for others, which hints to the fact that sampling is probably AI complete.

In our context, some inputs may lead to high compression ratios, while others do not. Consider the following function, for example: `f1(x1,x2,x3,x4) := insert(range(x1, 20), repeat(x2, [x3]), x4)` with the target `[N,N,...,N,0,0,...,0]` where a large number `N` is repeated 20 times and followed by zeros. Some values for `x1` leads to long input descriptions, for example, `x1=14` leads to `x2=6, x3=N, x4=[N,N,...,N,0,0,...,0]` after inverting the function, where the lange number `N` occurs 14 times within `x4` while lower values of `x1` lead to shorter input descriptions. Thus, the total description length of the inputs depends on `x1`: lower values of `x1` lead to more compression. Therefore, some gradient descent procedure would be helpful in order to find the minimum (`x1`=0, cutting out all numbers `N` from the target). However, a priori our algorithm would not perform a gradient descent procedure but exhaustive search by default, which is inefficient. Usually, people start hard-coding a special algorithm, like gradient descent in this case, in order to find the minimum more efficiently. Instead, we have followed our general paradigm of refraining from such heuristics and used our AIXI agent, in order to find the minimum.

The state space was set up as follows. There were two actions allowed, $+1$ and -1 to modify `x1`, no observations were needed for that task and rewards were set to $+1$ if the input description becomes shorter and -1 otherwise. The agent looked one time step ahead: $m = k$. The agent was initialized with a start value `x1=15`, an action history `-1,1,1,-1` with the respective rewards `1,-1,-1,1`, since decreasing `x1` leads better compression ratios. Given that history, the induction system of the agent has figured out that, among others, the function `f2(x) := map(negate, x)` is able to compute the reward sequence from the action sequence. That same function can then be used to compute future rewards for any action and thus to maximize them. In this way, the agent has figured out to take action `-1` at every time step. However, the agent was not able to figure out appropriate actions when the history had some noise, since filtering out noise currently exceeds the abilities of the induction system. Nevertheless, an effective gradient descent ability has emerged from the agent which relieves us from having to implement it as a heuristic. Moreover, nothing additional will have to be implemented in future, since AIXI is a generally intelligent agent and can be applied to a wide range of tasks. This is an example of self-improvement, since AIXI uses the induction system (it has found function `f2`) to improve the induction system (to find inputs to the function `f1` more efficiently). Note that even though from the perspective of intelligence, figuring out that reducing a number reduces some objective function is not a hard thing to do, the real achievement is about the fact that WILLIAM has found the solution on its own and has thereby drastically reduced the size of *its own* search space. This ability to recognize properties of a given task and deriving a task-dependent efficient procedure instead of blindly applying some general but inefficient one, is a crucial step for any system striving for AGI.

3.6 Intelligent Behavior: WILLIAM Plays Tic-tac-toe

Figure 3 shows a sequence of tic-tac-toe positions and alternate moves.

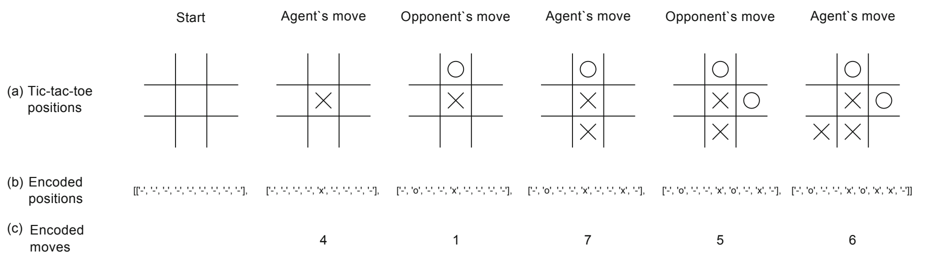

Start
Agent`s move
Opponent`s move
Agent`s move
Opponent`s move
Agent`s move
(a) Tic-tac-toe positions
(b) Encoded positions
[['-', '-', '-', '-', '-', '-', '-', '-', '-'],
['-', '-', '-', '-', 'x', '-', '-', '-', '-'],
['-', 'o', '-', '-', 'x', '-', '-', '-', '-'],
['-', 'o', '-', '-', 'x', '-', '-', 'x', '-'],
['-', 'o', '-', '-', 'x', 'o', '-', 'x', '-'],
['-', 'o', '-', '-', 'x', 'o', 'x', 'x', '-']]
(c) Encoded moves
4
1
7
5
6

Fig. 3. WILLIAM plays tic-tac-toe. (a) A sample game. (b) The compression target is the sequence of positions is encoded as a list of lists of strings. (c) The moves are encoded by the square number. The agent's last move on square 6 is a fork that cannot be defended – a smart move.

We encoded the sequence of positions into a list of lists, each of which contains 9 strings of length 1, where `'x'` denotes the agent's move, `'o'` the opponent's move and `'-'` a yet empty square. WILLIAM receives a target like Fig. 3(b) (except the last position) and tries to compress it.

Relatively quickly, the function `ttt := cumop3(setitem, START, MOVES, repeat(NUMREP, CONTENT))` with respective inputs.

`START=['-','-','-','-','-','-','-','-','-']`, `MOVES=[4,1,7,5]`, `NUMREP=2` and `CONTENT= ['x','o']` is found by the induction algorithm. `repeat(2, ['x','o'])` evaluates to `['x','o', 'x','o']`. `setitem` is an operator that writes some content in a list at the specified position. For example, `setitem(['a','b','c'], 2, 'x')` evaluates to `['a','b','x']`. Finally, `cumop3` is an arity 3 operator that cumulatively applies the function at its first input to the other inputs computing the list of encoded positions in Fig. 3(b).

We observe that the found function `ttt` is a tree of depth two which can be found quickly. The only thing that had to be programmed specifically for this game is the map from a position to the reward, since recognizing that a positive reward is given whenever there are three x-es in a row currently exceeds the induction abilities of the system. A winning position is rewarded by $+1$ and a losing one by -1. An illegal move is penalized by reward -100 if done by the agent and by $+100$ if done by the opponent.

The only thing that remains to be done by WILLIAM is to find a sequence of moves up that lead to a winning position. This kind of exhaustive search is already part of the generic AIXI implementation: the expectimax operation: it is not specific to tic-tac-toe. We chose the horizon of $m = k + 3$, i.e. 4 moves ahead (two taken by the agent, two by the opponent). However, instead of considering all permutation of all possible lists of four lists and compressing all possible futures, as would be required by AIXI, we used the found function `ttt` to run the prediction algorithm (Sect. 2.3) to *compute* the probable futures. Note that the sequence of past moves `[4,1,7,5]` was found by inversion. The generic prediction algorithm took the induced function and their inputs and extended it by looking at all continuations `[4,1,7,5,a,b,c,d]` of length 4 of a list of integers. The extended target contains either 4 possible future positions if $0 \leq a, b, c, d \leq 8$ or throws an index error (since `setitem` could not write on an index larger than the length of its list). The agent does not know that only numbers from 0 to 8 are valid moves. It simply attempts the numbers with the lowest description length first. Since the associated reward was given for free, the reward for every length-4-sequence of moves can be computed and the best move can be selected using the expectimax operation. Note, that minimax is a special case of expectimax, if the probability of the opponent making a move that minimizes the value of the agent's position, is set to 1 and the other moves to 0 (see [6], Chap. 6.4).

Note that apart from the reward computation, the agent doesn't know anything specific about the game. For example, if only the first starting position `['-','-','-','-','-','-','-','-','-']` is given, the agent compresses it simply with `repeat(9,['-'])`. Curiously, instead of making a `'x'`, WILLIAM's "move" is to attach `'-'` to this target, since this function is simpler than the `ttt` function above. After all, a list of dashes is all the agent has seen in its life. However, when more moves have been made, the agent finds a different description that successfully captures the regularities in the sequence of positions, namely, that each position is generated by the previous position by making an `'x'` or an `'o'`. The last move of the agent on square 6 in Fig. 3 is a fork that can not be defended. It is selected since no answer by the opponent can compromise the agent's victory.

4 Discussion

We have demonstrated the current state of WILLIAM, showing the ability to compress a diverse set of simple sequences, predict their continuation and use these abilities to solve simple tasks. As shown in Sects. 3.2 and 3.3, much larger program trees could be induced than would be tractable by exhaustive search, which consitutes evidence of the practical usefulness of our theory of incremental compression.

4.1 Related Work

The field of inductive programming has traditionally focused on the derivation/search of recursive or logic programs, which has generally suffered from the intractable vastness of possible programs to induce (see [8], [2] for a review), nurturing the demand for an incremental approach. The subfield of genetic programming can be viewed as one such approach making it beyond toy problems, since candidate programs are synthesized from already promising previous attempts or by small mutations. The problem of bloat however has challenged progress in the past [11]. We feel that algorithmic information theory can help out by providing theoretical guarantees for the implementation of such a challenging endeavor. The present paper can be viewed as an example of such a collaboration between theory and practice.

Another attempt to deal with the curse of dimensionality is to try to make the system itself deal with its own problems, by recursively improving itself. Adaptive Levin Search [12] is comparable to our approach being an inductive system with life-long self-improvement. It updates the probability distribution of the primitives in order to speed up the system based on acquired knowledge. WILLIAM is also able to do this, but moreover it can solve tasks incrementally, such that it does not need to find a difficult solution at once, but instead breaks down this process into steps.

4.2 Limitations

The list of current limitations is long both on the practical and theoretical side. Problematic is the aspect that the used language is not Turing complete, since the resulting Python programs always halt: there are no infinite loops and no recursion. We will change this by using loop operators but the search algorithm will have to be changed due to the halting problem, possibly using dovetailing or a yet to be developed computable theory of incremental compression. The conceptual problem of concatenating the leaves of a tree in the formation of the residual description is also problematic as presented in Sect. 3.2.

A systematic evaluation of the compression algorithm comparing it to the state of the art is also missing yet. Note that the tree in Fig. 1(a) is a way to cut out arbitrary repeating substrings from a string. For example, the widespread celebrated Lempel-Ziv algorithm also capitalizes on repeating substrings. We haven't tried it yet, but it seems straightforward to keep cutting out substring after substring during incremental compression using the very same function. This way, we can expect similar compression ratios for sequences at which Lempel-Ziv compression is good while possessing much more general compression abilities than specialized compression algorithms.

Another big missing piece is the lack of a dynamic memory. After all, pieces of programs that have proven useful, should be reused in future in order to enable incremental, open-ended learning. For this purpose, we could learn from successful memoization techniques in inductive programming, such as in [7]. On a more general note, the optimal structure of a memory is a difficult theoretical problem and to be embarked on in future.

4.3 Recursive Self-Improvement

It is interesting to observe that many of the current problems in this induction system could potentially be solved by the system itself. After all, it is a general problem solver when attached to AIXI. We have already demonstrated one self-aided way to search for inputs to a function in Sect. 3.5. Note that AIXI's problem solving abilities mostly depend on the abilities of the induction system. But if some version of the system uses AIXI to solve some of its own problems, it thereby effectively builds a new version of the induction system itself. Currently, WILLIAM is yet too weak to help itself on a large scale, but several other self-help problems come to mind.

For example, instead of using exhaustive search for various trees, we should bias the search toward more simple trees first. But simplicity is measured only after finding a short description, i.e. by finding a tree describing the tree! Hence, this would require a search through short trees that generate codes for other trees. The latter ones would then be simple by definition, since they have got a short description, and therefore more likely a priori, as given by the Solomonoff prior, Eq. 2.

Another example is the reuse of found trees by encapsulating them as composite operators, which is already possible. This way frequently used such composite operators will receive a short code (e.g. Huffman) increasing the likelihood of being reused.

Another issue is noticed for example in the prediction of the tic-tac-toe moves. The agent does not know anything about the game, it simply tries to extend a list of integers, such as the moves in Fig. 3(c). This leads to the attempt to try all integer combinations with the length defined by the AIXI horizon, including invalid combinations like `[0,0,0,2534]` which has a shorter description length than a set of valid moves `[6,8,7,5]` due to the logarithmic coding of natural numbers in the Elias delta code. A "smart" way would be to notice that any move above 8 makes the function throw an exception. Instead of building a heuristic, we plan to reuse WILLIAM's induction abilities in the framework of general knowledge-seeking agents [10], in order to find a function that computes the possible valid entries. In this case, a function like `lessthan(a, 9)` returns `True` for valid entries and its inversion would work as a generator for valid entries. Since the theory of knowledge-seeking agents already provides with optimal "experiments" that can rule out wrong hypotheses and WILLIAM is a system that comes up with those hypotheses, we can expect WILLIAM to be able to generate those inputs that are likely to be valid and help it to solve its own problems.

4.4 Compression, Interpretability and Concept Acquisition

Apart from facilitating the implementation of AIXI, one of the reasons we think that compression is important is the hypothesis that it facilitates the acquisition of concepts and reaches interpretable representations. Consider the rectangle example in Sect. 3.3. The compression target was a list of pixel coordinates which did not contain the width and length of the rectangle explicitly. Nevertheless, the final residual description did contain those variables. In fact, we suspect that if we run the algorithm on an ensemble of different rectangles, the residual will distill those variables even better, since those are the only unpredictable, i.e. incompressible, changes between rectangles. Therefore, the distillation of interpretable variables which could be mapped onto concepts and words for concepts is an important step toward building an agent endowed with conceptualizations tightly bound to grounded representations of the world. Moreover, for the purpose of building taxonomies of objects, deciding statements like "any square is a rectangle" appear possible since the (residual) description of a square is a special case of the (residual) description of a rectangle. This is a non-trivial observation since it is usually hard to obtain such taxonomic relationships in distributed representations such as in neural networks.

4.5 Training Time and Generalization Abilities

Speaking of neural networks, another striking difference between our approach and many common approaches in machine learning is that much less training is required in order to solve tasks. "Big data" is necessary exactly because many methods in machine learning do not generalize well. The lack of previous knowledge is not the only reason why so called one-shot learning is difficult for conventional methods. A major reason is that those methods do not compress data well. As exemplified in the tic-tac-toe example, the agent plays well in the very first game. Reasonable predictions in Table 1 are possible after the very first sequence seen by the algorithm. Moreover, the fact that compression leads to optimal generalization abilities is a fact proven in the theory of universal induction [13]. Therefore, heading for better compression ratios in machine learning is another message we would like to convey in this paper.

4.6 Could this be a Path Toward AGI?

In the face of the AGI challenge, the current results are very modest, to say the least, even though we don't see any fundamental limits to this approach, since it is backed by sound theories and any regularity seems to be representable by the current or future version of the algorithm. The scalability of any algorithm is usually impeded by the curse of dimensionality. In this case, our theory of incremental compression and the encouraging aspects of self-improvement emerging from the algorithm provide a fundamental response to this question, grounding the hope for scalability in future, although it is too early to say for sure.

In summary, we have demonstrated a general agent able to solve tasks in a range of diverse and simple environments. It searches for representations in a general algorithmic space instead of using a fixed representation usually employed by machine learning approaches or a patchwork of algorithms in cognitive architectures. Nevertheless, it achieves its relative efficiency by exploiting the fact, that our environment usually contains features that can be searched for incrementally. Possessing (nondegenerate) features is an assumption that is possibly not valid for the universal set of strings, however it may be valid for the universe we live in. In this sense, looking for such general but non-universal properties may boost the efficiency even further without making compromises on the generality of intelligence.

References

1. Elias, P.: Universal codeword sets and representations of the integers. IEEE Trans. Inf. Theor. **21**(2), 194–203 (1975)
2. Flener, P., Schmid, U.: An introduction to inductive programming. Artif. Intell. Rev. **29**(1), 45–62 (2008)
3. Franz, A.: Artificial general intelligence through recursive data compression and grounded reasoning: a position paper. arXiv preprint arXiv:1506.04366 (2015)
4. Franz, A.: Some theorems on incremental compression. In: Steunebrink, B., Wang, P., Goertzel, B. (eds.) AGI -2016. LNCS (LNAI), vol. 9782, pp. 74–83. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41649-6_8
5. Franz, A., Löffler, M., Antonenko, A., Gogulya, V., Zaslavskyi, D.: Introducing WILLIAM: a system for inductive inference based on the theory of incremental compression. In: International Conference on Computer Algebra and Information Technology (2018)
6. Hutter, M.: Universal Artificial Intelligence: Sequential Decisions based on Algorithmic Probability, p. 300. Springer, Berlin(2005). http://www.hutter1.net/ai/uaibook.htm
7. Katayama, S.: Towards human-level inductive functional programming. In: Bieger, J., Goertzel, B., Potapov, A. (eds.) AGI 2015. LNCS (LNAI), vol. 9205, pp. 111–120. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21365-1_12
8. Kitzelmann, E.: Inductive programming: a survey of program synthesis techniques. In: Schmid, U., Kitzelmann, E., Plasmeijer, R. (eds.) AAIP 2009. LNCS, vol. 5812, pp. 50–73. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11931-6_3
9. MacKay, D.J.C., Mac Kay, D.J.C.: Information Theory, Inference and Learning Algorithms. Cambridge University Press, Cambridge (2003)
10. Orseau, L., Lattimore, T., Hutter, M.: Universal knowledge-seeking agents for stochastic environments. In: Jain, S., Munos, R., Stephan, F., Zeugmann, T. (eds.) ALT 2013. LNCS (LNAI), vol. 8139, pp. 158–172. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40935-6_12
11. Poli, R., Langdon, W.B., McPhee, N.F., Koza, J.R.: A field guide to genetic programming. Lulu. com (2008)
12. Schmidhuber, J., Zhao, J., Wiering, M.: Shifting inductive bias with success-story algorithm, adaptive levin search, and incremental self-improvement. Mach. Learn. **28**(1), 105–130 (1997)
13. Solomonoff, R.: Complexity-based induction systems: comparisons and convergence theorems. IEEE Trans. Inf. Theor. **24**(4), 422–432 (1978)
14. Solomonoff, R.J.: A formal theory of inductive inference Part I. Inf. Control **7**(1), 1–22 (1964)