APPLICATIONS OF LEVIN'S UNIVERSAL OPTIMAL SEARCH ALGORITHM

Norbert Jankowski[1]

Department of Computer Methods
Nicholas Copernicus University
in Torun, Poland

Abstract

New applications of Levin's Universal Optimal Search are presented in this paper[2]. Using this method one finds solutions to a few chosen problems. Such solutions are characterized by possibility of the maximal generalization. In the deterministic version of the Universal Optimal Search algorithm one can always generate the best solution.

It is shown that it is possible to find a solution by this method for problems in neural networks for which back-propagation type of methods face difficulties.

Another application considered here is a problem of finding an exit-way from a maze.

There exists a probabilistic version of Levin's universal search [4]. This work offers an alternative implementation of the Levin's Universal Optimal Search.

Key words: Levin's complexity, Kt complexity, Universal Optimal Search, Kolmogorov complexity, Neural Networks.

1 Introduction

The algorithm of Levin's universal optimal search introduced by Levin in 70's [2], has been recently implemented by Jürgen H. Schmidhuber [4]. This implementation was based on random generation of programs. Here the deterministic version of the universal search is proposed.

[1]e-mail: norbert@phys.uni.torun.pl, http://class1.phys.uni.torun.pl/people/norbert.html

[2]You can obtain this paper on anonymous ftp server class1.phys.uni.torun.pl in directory /pub/papers/kmk/LUS-zakopaneV95.ps.Z.

To define universal search algorithm and its deterministic version we need a few formal definitions.

Let us start from the definition of Kolmogorov complexity [1].

DEFINITION 1 (KOLMOGOROV COMPLEXITY) Let the universal monotonic machine U scan an initial input segment p before it prints x (without necessarily halting after it does so). Let $l(p)$ be the length of program p. Then, Kolmogorov complexity is defined by

$$Kt_U(x) = \min_p\{l(p)\}. \tag{1}$$

The above definition does not take into account the time (number of steps) involved. Therefore a generalized definition of complexity was proposed by Levin [2] which consider not only length of a program but also logarithm of thetime of execution. For our research we used its probabilistic version [3].

DEFINITION 2 (CONDITIONAL LEVIN COMPLEXITY) Let us consider inversing problem ϕ and ϕ-witness x. Let the universal machine scan input segment p before it print w and let $t(p,x)$ be equal to number of time steps needed to print w. Then Kt' (conditional Levin's complexity) is defined by

$$Kt'_U(w|x,\phi) = \min_p\{l(p) + \log t(p,x) : \text{ program } p \text{ prints } w \text{ and test } \phi(w) = x \text{ in } t(p,x)\}. \tag{2}$$

Universal search is a way to solve arbitrary inverting problems in time which is optimal (see theorem 7.21 in [3]).

With this background we can describe *universal optimal search* algorithm: Levin's universal search algorithm will generate all strings-programs w in order of increasing

$Kt'(w|x,\phi)$ as long as $\phi(w) = x$ is true. More precisely Levin's universal search in phase i ($i = 1, \ldots, k$ where k is first such that $\phi(w) = x$) runs lexicographically all self-delimiting programs p (of length less than k) for $2^i 2^{-l(p)}$.

In next section we shall describe practical aspects of implementation more precisely. Special kind of Turing machine with one tape (which consists of work and program parts) will be used.

In section 3 and 4 we present applications of LUS in neural networks to find the simplest solution for given problem. Likewise solutions to another kind of problems are considered — finding an exit way from a maze. All applications show generalization property.

2 Deterministic version of Universal Optimal Search

The probabilistic version of universal optimal search has been recently proposed and implemented by Jürgen Schmidhuber [4]. It is based on the random generating of programs. Brief comparison between probabilistic and deterministic algorithms for universal optimal search will be given in the last part of this section.

The goal of deterministic version of Universal Optimal Search is generating (and checking) programs, starting with very simple programs and going successively to more complicated programs in the sense of Levin's complexity. This is possible because there are only finitely many programs which have Levin's complexity not bigger than some constant over a given set of *instructions* used to build programs (number of instructions is finite as well). Once all these programs are generated we know their Levin's complexity and we can sort them with respect to it.

Of course usually we don not know at the beginning the complexity of the actual problem (of the program generated by the LUS algorithm as the solution of this problem). One way to solve this is to generate in the first step programs of Levin's complexity not bigger than c_1, in the second step programs Levin complexity not bigger than c_2 etc, where $c_1, c_2, \ldots$ satisfy condition $c_1 < c_2 < \ldots$. We used $c_i = i$ because satisfying this condition the number of programs we generate is never larger than 2 times the number of programs of Levin's complexity smaller than that of considered the problem considered.

Tapes. Similarly to [4] we shall use single continoues Turing tape. The tape consists of finite sequence of cells. Every such cell contains an integer number between MIN_VALUE and MAX_VALUE and has an address in the interval $[-SW, SP]$. Cells of the interval $[-SW, -1]$ are cells of a *work tape* and cells of the interval $[0, SP]$ are cells of a *program tape*. Cells of program tape are read-only, and cells of work tape are read-write.

The current program is located between 0 and Max whereas the work tape between Min and -1, during execution of generated programs (if Max is equal to -1 there is no program on the program tape). Before any program will use a work tape it must allocate needed quantity of cells (later this work space may be freed by the program).

Instructions. As mentioned above the tape consists of integer numbers. Each one of these numbers denotes either instruction number (below we shall call it primitive) or argument or value of *pure work* cell.

A set of primitives (instructions) is used. Primitive is bound with its number between 0 and $NR\_OF\_PRIMITIVES-1$. It is possible that primitive has arguments, they are placed directly behind a given primitive. Table 1 shows a set of primitives used throughout this work.

How does deterministic version of the universal search work? Deterministic version executes successively parts $\mathcal{P}_i$ ($\mathcal{P}_1, \mathcal{P}_2, \ldots$).

A given part $\mathcal{P}_i$ generates and checks all programs of complexity not larger than i. The description of how does each part $\mathcal{P}_i$ work is presented below.

❶ At the very beginning work tape is empty ($Min = -1$), program tape is empty ($Max = 0$) and $InstructionPointer$ is set to zero.

❷ One primitive – cell is added at the end of a program tape (if added primitive uses any arguments the argument cell is also added). The primitive which is added must satisfy the following condition: its number is the smallest one such that the complexity of *changed* program obtained this wayis still not bigger than i. If there is no such primitive then go to step ❹.

❸ A primitive that the variable $Instruction Pointer$ pointed is ran. Then the $InstructionPointer$ is actualized according to the kind of primitive and its result. If result of execution of primitive is either $STOP$ or $ERROR\_\ldots$ or if complexity of the current program is exceeded then go to step ❹. If $InstructionPointer$ is equal to $Max+1$ then go to ❷, otherwise step ❸ is executed again.

❹ Program from the work tape is checked (in practice this step is reduced to checking the result of the program). Suitable result is saved and/or displayed.

❺ Next program (or its prefix) is created on the ground of the last checked program (for more information see next paragraph). If it ends successfully then $InstructionPointer$ and Min are initialized with zero and step ❸ executed, otherwise it means that all programs of complexity i were already generated and checked and the $\mathcal{P}_i$ part is finished.

Creating next program. When the program from the program tape is stoped (due to time limits or excessive length) during the universal search next program or its

Table 1: Universal set of primitives.

Number	Name	Description
0	jumpleq(*arg1*, *arg2*, *arg3*)	jump to *arg3* if only the content of address *arg1* is not bigger than the content of *arg2*
1	output-weight(*arg1*)	it puts the content of address *arg1* to weight $w_{WeightPointer}$ and variable $WeightPointer$ is incremented
2	jump(*arg1*)	$InstructionPointer$ is set to the content of address *arg1*
3	stop	stops program
4	add(*arg1*, *arg2*, *agr3*)	add the content of address *arg1* and the content of address *arg2*, result is putting in address *arg3*
5	get input	it is not used
6	copy(*arg1*, *arg2*)	the content of address *arg1* is copied to address *arg2*
7	allocate(*arg1*)	it allocates *arg1* cells on work tape
8	increment(*arg1*)	the content of address *arg1* is incremented
9	decrement(*arg1*)	the content of address *arg1* is decremented
10	subtract(*arg1*, *arg2*, *arg3*)	the content of address *arg1* is subtracted from content of address *arg2*, result is put into address *arg3*
11	multiply(*arg1*, *arg2*, *arg3*)	the content of address *arg1* is multiplied by content of address *arg2*, result is put into address *arg3*
12	free(*arg1*)	it frees *arg1* cells from work tape

Table 2: Changes of previous set of primitives.

Number	Name	Description
1	write weight (arg1, arg2)	weight w_i is set up to the content of address *arg1*, where i is equal to the content of address *arg2*
5	read weight(arg1, arg2)	to cell at which the address *arg1* points put the value of weight w_i is equal to the content of address *arg2*

Table 3: Primitives for finding the exit way from maze.

Number	Name	Description
0	TurnRight	turn right
1	TurnLeft	turn left
2	GoAndBreak(addr)	if it is possible go in current direction, then go, otherwise jump to address *addr*
3	GoAndJump(addr)	if it is possible go in current direction, then go and jump to address *addr*, otherwise do nothing

prefix should be generated. As we have described above the last program cell and its argument cells (sometimes none) are placed on the left of the cell pointed by a variable Max.

At the beginning of a building of the next program universal search tries to increment the last argument. If it is possible the program or its prefix is generated. If not universal search tries to increment previous argument (if any exists) and so on. If no arguments left the content of the primitive-cell is incremented and on success next program is generated. If complexity of a program obtained in such way is larger than currnet considered complexity then will be repeated incrementation of the content of the primitive-cell.

If all the above incremental-trials fail then value Max is set to address of the last primitive-cell (it is equivalent to removal of the last primitive with its arguments from the program tape). Then the whole procedure is repeated (if Max is larger or equal to zero, otherwise we can not generate another program of a given complexity).

Universality. As it is described in [4] set of primitives shown in table 1 is universal — it enables generating of *"any computable integer sequence onto the work tape (within given size and range limitation)"*. Primitives used in section 4 (see figure 3) for finding exit-way from a maze form a universal set of primitives as well: it is possible to generate (find) any *exit way program* for a given maze.

Brief summary of differences between deterministic and probabilistic versions of universal optimal search

Deterministic version:

☛ this method generates sequentially programs according to their complexity

☛ programs found by deterministic version are better than programs found by probabilistic version (more precisely they are never worse). One could say that these programs may be regarded as having more regularities or generalizing better

Probabilistic version:

- ☞ next program is generated in a random way
- ☞ programs found by probabilistic version often are 'wild'
- ☞ solution is usually found a bit quicker (because deterministic version **must** consider all programs which complexity is lower than complexity of curnet problem before it found first solution)

3 Towards Neural Networks

In this section we shall consider examples considered by Jürgen Schmidhuber in [4] in his probabilistic version of universal search. We shall present below solutions found by deterministic version of universal search.

Similarly to Jürgen Schmidhuber's papers it will be shown that one can find weights for neural network for which standard methods e.g. backpropagation fails.

3.1 Counting perceptron

Definition of the task: Our goal is to find a neural network that counts number of 1's on input units. Each of 100 input units x_i can be equal to either 1 or 0. To solve this problem a network with 100 input units $x_0, \ldots, x_{99}$, without any hidden unit and with one output unit will be used. Global output is equal to

$$\mathbf{y}^p = \sum_{i=0}^{99} x_i^p w_i \qquad (3)$$

where w_i is i-th weight of neural network.

We should find satisfactory set of weights for this neural network. There exist only one solution given by $w_i = 1$ for each i. In such a case the output $\mathbf{y}^p$ is equal to number of 1's on the input. Every input pattern has precisely three 1's and ninety seven 0's.

We shall use 3 training examples. This may be quite challenging for neural networks, for example the backpropagation algorithm usually fails in this situation. Our training examples are: first vector x^1 with 1's at positions 5, 17 and 86, second vector x^2 with 1's at positions 13, 55 and 58 and third vector x^3 with 1's at positions 40, 87 and 94. According to the above definition outputs for vectors x_1, x_2, x_3 are equal to 3.

Results: First solution was found after 13,328 runs. It is presented in table 4. Program used 194 time steps, its space probability (i.e. probability of selection of this program) is 0.000739645 and complexity is equal to 18.0008. This vector fits 147,440 of all 161,700 examples.

Notice that in this case weights don't fit all examples because the time necessary to find all solutions was bigger than that allowed by current complexity. Such program we shall call *acceptable program*. In this case to find a *good program* (fitting all examples) it is sufficient

Table 4: Counting perceptron — first program found.

Address	Content	Command/Argument
0	1	output-weight
1	0	0
2	2	jump
3	0	0

Address	Interpretation
0	Write the content of address 0 (it is 1) to weight pointed by $WeightPointer$, increment $WeightPointer$, if $WeightPointer$ is equal to 100 then program is stopped.
2	jump to address 0

to increase complexity by small constant (e.g. 1). Adequate program was found after 21,180 runs, it used 199 time steps, and its total complexity was 18.0375. So, the first solution was very close to that one.

Table 5: Counting perceptron — a little faster solution.

Address	Content	Command/Argument
0	1	output-weight
1	0	0
2	1	output-weight
3	0	0
4	2	jump
5	0	0

Slightly faster program was found with complexity 23.9054 (faster solution is more expensive). It used 148 time units and its space probability was 9.48263e-06. Code of this program is presented in table 5. This was the 77th program fitting all examples.

Generating 100,000 programs we have found 8 good programs and 2 acceptable programs (but both are really good programs — these programs had no possibilities to use needed time). For more information see section 5.

3.2 Adding perceptron

Definition of the task: This problem is a bit similar to the previous one. We have a network with 100 input units and one output unit. Output of the network as before is counted out by the formula (3). But now our goal is to find a network (weights for a network) which has on the output $\mathbf{y}^p$:

$$\sum_{\substack{i=0 \\ x_i=1}}^{99} i$$

Training vectors (and their number) are the same. The target has been changed according to the following definition: vector $x^1 = 108$, $x^2 = 126$ and $x^3 = 221$. Solution for the problem considered is such that each weight w_i is equal to i.

The results: First vector of weights was found after 34,672,685 runs. This program fits correctly all examples. It used 399 time units and allocated 100 cells on the work tape, but using in practice only 1 cell (the first one on the work tape — address -1). Space probability was $5.55758 * 10^{-8}$ and complexity of this program was 32.7412.

As one can see Levin's universal search has very nice generalization property. It **always** uses the main regularities that are intrinsic attributes of the searched object and (what is very important!) it is independent (up to an additive constant in complexity) of the set of used primitives (in other word, it is sufficient if set of primitives is universal).

Second program is more frugal of the work tape and time. It allocates only one additional cell on the work tape (see table 6).

Table 6: Adding perceptron — more economical solution (content of work tape after execution).

Address	Content	Command/Argument
-1	100	100
0	7	allocate
1	1	1
2	8	increment
3	-1	-1
4	1	output-weight
5	-1	-1
6	2	jump
7	2	2

This program used 300 time units. The space probability was exactly the same, and complexity was only 32.3298. Notice that if the complexity of two programs differs by less than 1 then it is possible that program with the slightly larger complexity will be found before the second one. Such solution was found after 34,672,688 runs.

29 programs (28 fitting all examples) were found after $4.65*10^8$ runs. The average time of running was 120.756.

3.3 *Counting and Adding perceptron with another set of primitives*

To obtain another set of primitives used in the next two experiments prmitives number 1 and 5 have been changed (see table 2). The variable $WeightPointer$ is now automaticaly incremented. The program stops when it meets primitive *stop* or when it tries to make an inadmissible action (for instance *jump* to non existing address and etc.).

After all these changes complexity of previous experiments should change only by a constant number independly of the problem considered, as *Invariance Theorem* [3] says.

The solution for counting perceptron. First solution was found by the universal search after $74, 526, 154$ runs. However this solution fits $134, 044$ of all $161, 700$ examples and it will never fit all examples because first weight w_1 was set to 7. This program used 477 time units, its space complexity was $2.77879 * 10^8$ and complexity 33.9988.

Second program found (with complexity 33.997) was good but was stoped too early and did not come in time to set all weights. Third program found fitted all examples and it had minimal complexity 33.6072. It used only 303 time units and its space probability was $2.31566 * 10^{-8}$. The corresponding program is shown in table 7. This program was found after 81,139,383 generated programs and averaged time necessary to run one of these programs was 94.2461.

Table 7: Counting perceptron with second set of primitives — third found program.

Address	Content	Command/Argument
-1	101	101
0	7	allocate
1	1	1
2	8	increment
3	-1	-1
4	1	write weight
5	-1	-1
6	1	1
7	2	jump
8	2	2

Table 8: Adding perceptron with second set of primitives — the best found program.

Address	Content	Command/Argument
-1	101	101
0	7	allocate
1	1	1
2	8	increment
3	-1	-1
4	1	write weight
5	-1	-1
6	-1	-1
7	2	jump
8	2	2

The solution for adding perceptron. Two solutions were found within 10^8 runs. The two programs are correct and fit all examples. After $3 * 10^9$ runs universal search found 11 good programs and one acceptable program. Second solution (see table 8) was better, it used 303 time steps and 1 cell on work tape. Its complexity was 35.4961.

4 Looking for an exit-way from the maze

Definition of the task: Below we present solution for the problem of finding an exit-way from mazes. In other words we shall look for the program which *will be able* to get out from the maze. We would like to stress that we are interested in the best algorithm for going out from a maze but not in the best scheme of finding such an exit-way. An optimal exit-way algorithm should use all regularities that occur in the maze though they might not arise from a set of primitives.

Another set of primitives will be used here. It consists of 4 primitives only (see table 3). Of course it is possible to enlarge it by a few primitives from one of the previous sets of primitives used for neural network problems or by some other primitives to obtain more flexible set which could make to building of the self-sizing programs possible.

Solutions. It may be seen from figures 1, 2, 3 that dimensions of mazes are not important, important is only how much a given maze is regular, what “complexity of this maze” is. Dimension of maze ($d_1 \times d_2$) determines only the minimal complexity which is $\log_2(d_1+d_2)$ (provided that *entrance* and *exit* are placed on oposite ends of a diagonal of the maze).

Figure 1 concerns first example[3]. Maze dimension is 7×7. The table with the best solution is placed on the left of maze–picture. The optimal exit-way algorithm needs 137 steps to get out from the maze (program’s time steps, not step as a move in the maze) and its space probability was $2.79 * 10^{-5}$ whereas complexity was equal to 22.2273.

Comparing to the next examples one notices that complexity of this example is quite big taking into account its dimension (7×7).

Another very interesting situation may be observed in figures 2 and 3. The two mazes are very similar. In the first maze was chosen sequential a whole in each vertical wall, in the second it was chosen randomly, but the difference of their complexity is quite important.

Notice that the relatively large maze from figure 2 has complexity which is almost equal to the minimal one amongthe mazes presented here.

5 Conclusions

We have presented first implementation of deterministic version of the Levin’s universal search algorithm.

Levin universal search is a good way of finding solutions for different kind of inversing problems, especially for inversing for which solutions obtained by another methods are not acceptable or are impossible to obtain.

[3]You can find an exit way represented by a black line between *entry* and *exit*, red color signifies walls of a maze (in print is grey)

Another magnificent feature of LUS is that if the *input* for a given problem is given but the method of solving the problem is unknow one can still find solution. We need only function ϕ to try whether the result is acceptable or not.

The main disadvantage of Levin’s universal search is its combinatorial explosion (when we look for solutions of problem with complexity bigger by one, we need two times more time).

During search for the solution of the counting perceptron problem presented in section 3.1 intermediate results were accumulated. The results are presented in figure 4.

The logarithms of number of *good* and *accepted* programs with different complexities are shown there. For a given complexity k only programs of complexity belongings to the interval $[k, k+1)$ are counted.

The most important is that programs–solutions are generated from the simplest in the direction to more complicated, as it was described in section 2.

These results show clearly that while searching space growths, the difference between probability of choosing *accepted* and *good* remains unchanged. It means that larger search spaces are not more *chaotic* as well as it is not more difficult to draw *good* or *accepted* program.

6 Acknowledgement

This researche was performed at Technical University of Munich during cooperation with Jürgen H. Schmidhuber, who is originator of the idea. I would like to thank him for his hospitality.

References

[1] A. N. Kolmogorov. Three approaches to the quantitative definition of information. *Prob. Inf. Trans.*, 1:1–7, 1965.

[2] L. A. Levin. Universal sequential search problems. *Problems of Information Transmission (translated from Problemy Peredachi Informatsii (Russian))*, 9, 1973.

[3] M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications*. Text and Monographs in Computer Science. Springer-Verlag, 1993.

[4] J. H. Schmidhuber. Discovering problem solutions with low Kolmogorov complexity and high generalization capability. Technical Report FKI-194-94, Fakultät für Informatik, Technische Universität München, 1994.

Figure 1: Maze I. Dimension: 7×7. Complexity: 22.2273. Time used: 137. Space probability: $2.79 * 10^{-5}$

Address	Contents	Commands/Arguments
0	0	TurnRight
1	0	TurnRight
2	1	TurnLeft
3	3	GoAndJump(addr)
4	0	0
5	2	GoAndBreak(addr)
6	2	2

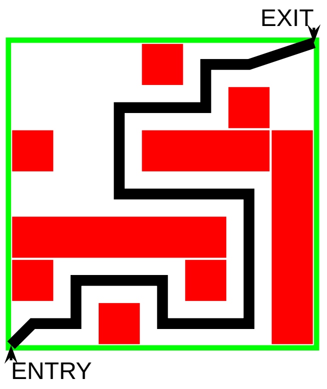


Figure 2: Maze II. Dimension: 100×100. Complexity: 21.1212. Time used: 495. Space probability: 0.000217014

Address	Contents	Commands/Arguments
0	0	TurnRight
1	3	GoAndJump(addr)
2	1	1
3	1	TurnLeft
4	3	GoAndJump(addr)
5	0	0

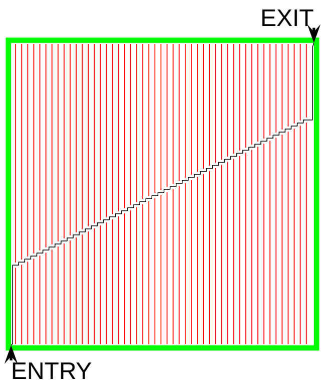


Figure 3: Maze III. Dimension: 100×100. Complexity: 29.5678. Time used: 29605. Space probability: 29.5678

Address	Contents	Commands/Arguments
0	1	TurnLeft
1	3	GoAndJump(addr)
2	0	0
3	3	GoAndJump(addr)
4	0	0
5	2	GoAndBreak(addr)
6	2	0

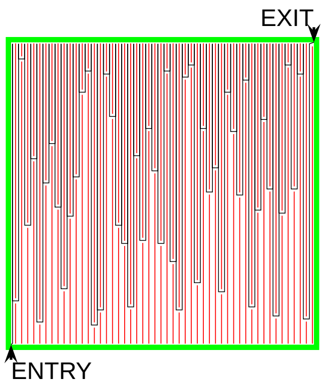
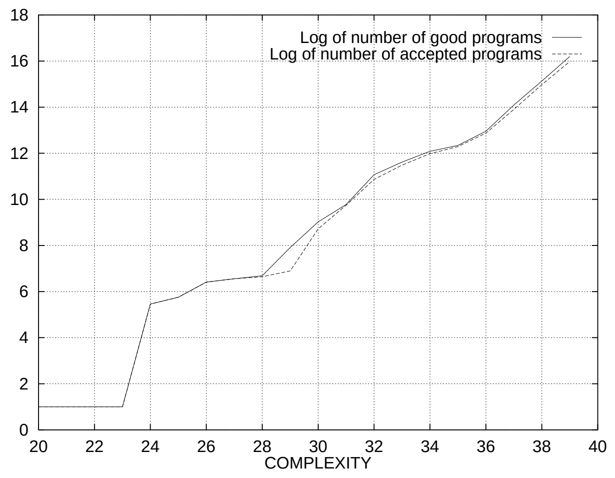


Figure 4: Similar growth of *good* and *accepted* program in growth of the complexity


Log of number of good programs
Log of number of accepted programs
COMPLEXITY