Probable General Intelligence algorithm

Anton Venglovskiy[0000-0001-5780-6572]

Kyiv, Ukraine, 2019

anton.venglovskiy@gmail.com

Abstract. This article contains a description of a generalized and constructive formal model for the processes of subjective and creative thinking. According to the author, the algorithm presented in the article is capable of real and arbitrarily complex thinking and is potentially able to report on the presence of consciousness.

Keywords: artificial general intelligence, algorithm of mind, self-organizing algorithmic chaos.

1 Introduction

In general, the author relies on the assumption that the logic of the phenomenon of mind is completely reduced to the logic of the phenomenon of constructive complexity. Simply put, reasonableness and complexity are one and the same thing and thinking mean sufficiently complex constructive behaving. Accordingly such phenomena as "understanding the meaning of things" and "problem-solving", the author considers as epiphenomenons in processes with more fundamental logic, which is a logic of constructive complexity, that goes beyond these particular phenomena.

In turn, the logic of constructive complexity can be expressed formally and build an algorithm. This algorithm allows to unlimited create unique content of any given constructive complexity, in a formal computing process, as a consequence of this, the algorithm is capable of simulating arbitrarily complex constructive behavior in dynamics. The author suggests that if in some computing process, possible to simulate arbitrarily complex constructive behavior, then this process is thinking. Thinking algorithms will think subjectively if their computational process is closed.

From a purely technical point of view, the process of thinking is self-organizing algorithmic chaos, which in the process of computing is able to spontaneously reach any complexity of its structural organization.

2 The logic of constructive complexity

The objects of the logic are abstract theories. Any theory has complexity and this complexity can be explicitly verified. From any theory, it is possible to deduce more complex theories. From any complex theory, simple theories can be derived. For different theories, inference will be different.

Abstract theories are any things about which it is known only that they are inherent in constructive complexity because this complexity can be explicitly verified. And it is also known that from any such thing a closed constructive transition to other, more complex things is possible and this can also be checked.

Constructive complexity. A complex object is something that can be decomposed into prime objects. The more prime objects contained in a complex object, the more complex this object. Prime objects cannot be decomposed. The complexity of all prime objects is the same.

In accordance with the above definition of complexity, abstract theories are divided into two types: prime and complex. A theory is called complex if using some procedure, a set of prime theories can be derived from it. In turn, using the same procedure for prime theories returns a constant result. The complexity of all prime theories is the same. Due to the fact that the concept of complexity in the logic under consideration is defined constructively, it can be calculated and compared. Two theories have the same complexity if they can be decomposed into the same number of prime theories. The more prime theories you can get from a complex theory, the more complicated the original theory.

The logic described above can be expressed in formal operations on strings of a special kind.

Set of abstract theories S. To represent theories, strings are used that consist of an arbitrary sequence of parentheses '(', ')' and any identifiers inside the parentheses. For brevity of explanation, further, each letter of the alphabet is considered a separate identifier. Identifiers can be repeated; each occurrence of the identifier is an independent syntactic unit. The entire contents of the string must be enclosed in common outer parentheses. The hierarchy of parentheses in the string is arbitrary, but there must be a closing one for each opening parentheses. Each correct string defines a tree. Example: string ((b)a(e)) is correct, while strings (b)a(e), (a(b(e) are incorrect. Another examples of correct strings: () ≡ ∅ - empty string, (a), (aa), ((aa)(aa)), (bb(b(aaa))(abb)). Two strings are considered identical if the trees corresponding to them are isomorphic. An example of how you can rearrange the elements of string: (ab(cd)) ≡ ((cd)ab) ≡ (b(dc)a) ≡ ... ≡ ((dc)ba). Any permutations are permissible if that does not change the tree of the string. Empty substrings are not significant and are thrown away, for example, (a()) ≡ (a). To reduce writing, repeated items can be written using a repeat prefix, for example: (aa) ≡ (2a), (aaabbb) ≡ (3a3b), (aa(bb)) ≡ (2a(2b)), ((a)(a)(a)) ≡ (3(a)), (aaa(aabb)(aabb)) ≡ (3a2(2a2b)).

Set **S** consists of all possible correct strings. On the set **S** three rules of inference are defined.

“Abstraction” rule. Applies to substrings of a source string. Allows you to put out from the parentheses (hereinafter PFP) the same content. From any group of parentheses at the same level, any identical substrings can be taken out of parentheses, according to the following principle:

$$((a)(b)) \Rightarrow \emptyset;$$

$$((a)(a)) \Rightarrow (a()) \equiv (a);$$

$$((ab)(ac)) \Rightarrow (a(bc));$$

$$((aa)(aa)) \Rightarrow \{(a(aa)), (aa)\};$$

$$((ab)(ab)) \Rightarrow \{(a(bb)), (b(aa)), (ab)\};$$

$$((a(b))(a(b))) \Rightarrow \{(a((b)(b))), ((b)(aa)), (a(b))\};$$

$$((ab)(abc)) \Rightarrow \{(a(bbc)), (b(aac)), (ab(c))\};$$

$$((ab)(ac)(ae)) \Rightarrow \{(a(bce)), (a(bc)(ae)), (a(ab)(ce))\};$$

$$((ab)(ac)(fe)(fk)) \Rightarrow \{(a(bc)(fe)(fk)), (f(ek)(ab)(ac))\};$$

Applying the “abstraction” rule to the source string, in the general case, a lot of resulting strings can be inferred. By the “abstraction” rule, results are always simpler than the source string. In the case of prime strings, the result of applying the “abstraction” rule is empty. The recursive application of the “abstraction” rule allows you to decompose any complex string into prime ones.

A more detailed example of the “abstraction” rule is given in the next section.

“Deduction” rule. According to this rule, from the source string you can get as many fundamentally new strings as you like, by duplicating all the elements in the source string any given number of times, according to the following principle:

$$(a) \Rightarrow \{((aa)(aa)), (3(3a)), (4(4a)), \ldots\};$$

$$((a)) \Rightarrow \{(((aa)(aa))((aa)(aa))), (3(3(3a))), (4(4(4a))), ...\};$$

$$(a(b)) \Rightarrow \{((aa(bb)(bb))(aa(bb)(bb))), (3(3a3(3b))), (4(4a4(4b))), \ldots\};$$

$$(a(b(cc))) \Rightarrow$$
$$\{(aa(bb(cccc)(cccc))(bb(cccc)(cccc)))(aa(bb(cccc)(cccc))(bb(cccc)(cccc))),$$
$$(3(3a3(3b3(6c)))), (4(4a4(4b4(8c)))), \ldots\};$$

By the “deduction” rule, from any source string, a fundamentally new and guaranteed more complicated string can be deduced and this fact can be checked using “abstraction” rule.

“Composition” rule. Any set of strings from S can be combined into one string. For example: (a), (b), (e) ⇒ ((a)(b)(e)).

Thus, a formal system is obtained which satisfies the definition of complexity logic. Within the framework of the described logic, it is possible to construct an algorithm that corresponds to the criteria of the subjective thinking algorithm.

3 Algorithm of General Intelligence

The algorithm is a recursive function of the following form:

$$t_n = (A[D[t_{n-1}]]);\ t_n \in \mathbf{S}$$

The function t_n produces algorithmically random and structurally unique content from any nonempty seed string $t_0 \in \mathbf{S}$. Content that produced in this way spontaneously organizes itself and its dynamic behavior can be arbitrarily complex. The potential amount and constructive complexity of such content are boundless. The calculation of this function is a process of thinking.

"Abstraction" operator A. This operator can be applied to any string from S. The result of applying the operator to the source string is a set of strings that contains all possible strings which can be obtained by recursively applying the "abstraction" rule to the source string. The recursion continues until all possible prime strings are obtained.

Let us consider step by step several examples of the action of A. Let the string be given ((aa)(aab)(aab)), this string has three substrings located on the same level: (aa), (aab), (aab), all three substrings have the same content fragments. For a PFP operation, we can arbitrarily select any combination of substrings. In this example, there are three different possible combinations of substrings: (aa) (aab); (aab) (aab); (aa) (aab) (aab). For each presented combination of substrings, all possible variants of PFP operation will be created. Step by step PFP for combination (aa) (aab):

first case:

- Choose content ((a<u>a</u>)(<u>a</u>ab) (aab)).
- PFP (<u>a</u>(<u>a</u>)(<u>a</u>b) (aab)).
- Merge (<u>a</u>(<u>a</u> <u>a</u>b) (aab)).
- Result (a(aab)(aab)).

second case:

- Choose content ((<u>aa</u>)(<u>aa</u>b) (aab)).
- PFP (<u>aa</u>(_)(_b) (aab)).
- Merge (<u>aa</u>(__b) (aab)).
- Result (a(b)(aab)).

PFP's for combination (aab) (aab):

- ((aa)(<u>a</u>ab)(<u>a</u>ab)) ⇒ (a(aa)(aabb)).
- ((aa)(<u>aa</u>b)(<u>aa</u>b)) ⇒ (aa(aa)(bb)).

- ((aa)(aab)(aab)) ⇒ (ab(aa)(aa)).
- ((aa)(aab)(aab)) ⇒ (b(aa)(aaaa)).
- ((aa)(aab)(aab)) ⇒ (aab(aa)).

PFP's for combination (aa) (aab) (aab):

- ((aa)(aab)(aab)) ⇒ (a(aaabb)).
- ((aa)(aab)(aab)) ⇒ (aa(bb)).

As can be seen from the above example, for the source string ((aa)(aab)(aab)) there are nine different cases to putting something out from the parentheses and there are nine resulting strings for these cases. It's not all possible PFP cases, but only nine pieces of the first iteration. It is also necessary to construct all PFP cases for each of the nine previously obtained results:

- (a(aab)(aab)):
  - (a(aab)(aab)) ⇒ (aa(aabb)).
  - (a(aab)(aab)) ⇒ (aaa(bb)).
  - (a(aab)(aab)) ⇒ (aab(aa)).
  - (a(aab)(aab)) ⇒ (aaab).
  - (a(aab)(aab)) ⇒ (ab(aaaa)).
- (a(b)(aab)):
  - (a(b)(aab)) ⇒ (ab(aa)).
- (a(aa)(aabb)):
  - (a(aa)(aabb)) ⇒ (aa(aabb)).
  - (a(aa)(aabb)) ⇒ (aaa(bb)).
- (aa(aa)(bb)).
- (ab(aa)(aa)):
  - (ab(aa)(aa)) ⇒ (aab(aa)).
  - (ab(aa)(aa)) ⇒ (aaab).
- (b(aa)(aaaa)):
  - (b(aa)(aaaa)) ⇒ (ab(aaaa)).
  - (b(aa)(aaaa)) ⇒ (aab(aa)).
- (aab(aa)).
- (a(aaabb)).
- (aa(bb)).

So,

A[((aa)(aab)(aab))] = {(a(aab)(aab)), (aa(aabb)), (aaa(bb)), (aab(aa)), (aaab), (ab(aaaa)), (a(b)(aab)), (ab(aa)), (a(aa)(aabb)), (aa(aabb)), (aaa(bb)), (aa(aa)(bb)), (ab(aa)(aa)), (aab(aa)), (aaab), (b(aa)(aaaa)), (ab(aaaa)), (aab(aa)), (aab(aa)), (a(aaabb)), (aa(bb))};

Consider a few more examples.

A[((a)(a(b)(b)))] =

- ((a)(a(b)(b))) ⇒ ((a)(ab)):
  — ((a)(ab)) ⇒ (a(b)).
- ((a)(a(b)(b))) ⇒ (a((b)(b))):
  — (a((b)(b))) ⇒ (a(b)).

A[((a(b))(a(b)))] =

- ((a(b))(a(b))) ⇒ (a((b)(b))):
  — (a((b)(b))) ⇒ (a(b)).
- ((a(b))(a(b))) ⇒ ((b)(aa)).
- ((a(b))(a(b))) ⇒ (a(b)).

"Deduction" operator D. A "deduction" rule with a fixed duplication parameter corresponds to the action of the operator D. For the nearest practical purposes, it is enough that the duplication parameter is 2. As a result of D[s] execution, all components of the source string s are doubled. Examples:

D[(a)] = (2(2a)) = ((aa)(aa));

D[(aa)] = (2(4a)) = ((aaaa)(aaaa));

D[(ab)] = (2(2a2b)) = ((aabb)(aabb));

D[(a(b))] = (2(2a2(2b))) = ((aa(bb)(bb))(aa(bb)(bb)));

D[((a)(b))] = (2(2(2a)2(2b))) = (((aa)(aa)(bb)(bb))((aa)(aa)(bb)(bb)));

D[((a)(b(cc)))] = (2(2(2a)2(2b2(4c))));

Composition Operator (). Corresponds to the action of the "composition" rule.

Substantive interpretation of "deduction" and "abstraction" operators. The physical meaning of the "deduction" operator is as follows. "Deduction" "blindly" adds qualitatively new information to any original object in a closed way and thereby produces a fundamentally new object that is necessarily more complex than the original object. In turn, the "abstraction" operator decomposes the new object into its components and, thus, constructively expresses the information added at the "deduction" stage. You may notice that when performing the PFP there is a loss of information. Roughly speaking, for this syntax, PFP operation is a universal way to meaningfully lose information in the absence of any a priori data about the meaning of strings. From the point of view of the algorithm, all possible variants of information loss, which are calculated at the stage of "abstraction", are, in fact, the value of strings. Thus, at each iteration, the algorithm produces a new and unique syntactic heuristic. And each of the following heuristics is fundamentally more complex and more informative than the previous one. At each iteration of the algorithm, fundamentally new knowledge spontaneously arises.

4 Practical application of the algorithm

In order to solve practical problems using the algorithm, it is necessary to rationally interact with it in the process of computing, as with an unknown intelligent object.

Consider the ideal case of interaction with the algorithm. Suppose we have unlimited processing power. Four essential elements are required for interaction:

1. Digital model of an interactive environment, the meaning of which is known.
2. Digital model of the tool that can affect the environment.
3. Encoding algorithm that will encode the state of the environment using strings from **S**.
4. Decoding algorithm, which will be in some fuzzy, but rationally motivated way to decode strings from **S** and convert them into signals for the tool.

Below is schematic diagram of interaction (in pseudo-code):

```
S NextThought(S prevThought, S ExternalSignal, int expo-
sure = 1)
{
   S t = (prevThought, ExternalSignal); //composition
   for (int i = 0; i < exposure; i++)
      t = (A[D[t]]); //thinking
   return t;
}

EnvironmentModel e;
S s = encode(e.GetState());
S o = Ø;
while (thinks)
{
   o = NextThought(o, s);
   e.ImpactTool.perform(decode(o));
   s = encode(e.GetState());
}
```

To optimize the calculations, you can use many short strings in parallel, and control the growth of their length in two natural ways:

1. Cut out content that is located in parentheses of deep nesting, because it is more chaotic than content located shallow. Shallow content is highly organized because it has “surfaced” from the depths as a result of iterative abstractions.
2. Carry out rational selection and grouping of strings at the stage of composition. For example, select only prime strings. In a general way, you can select and grouping strings based on preferred statistical properties.

5 The concept of the meaning of things in the logic of complexity

Any thing can be defined and meaningful only in a constructive relation to other things. In this paradigm, the meaning of things is defined in a potential and continuous way. This means that a hypothetical record of one's own exhaustive definition for any thing will be infinite and infinitely complex. In the logic of the complexity of each thing, corresponds is a certain syntactic representation, and the meaning of the thing is a potentially possible syntax, to which a constructive transition from actual syntax can be made. If the amount and complexity of the potential syntax are infinite, then the source entry is informative and corresponds to some meaningful thing. Each entry in the logic of complexity is informative. From the indicated positions, the t_n function can be considered as a process of unrolling the inner meaning of things, which corresponds to an intuitive idea of the thinking process.