An Improved Generalized Enumeration of Substitution Systems

Kenneth E. Caviness
Camille Morrow
Christen Case
Victoria Kratzke

Physics and Engineering Department
Southern Adventist University
Collegedale, TN 37315
caviness@southern.edu

The enumeration of all sequential substitution system rulesets is modified to include generalized substitution system rulesets. Unlike its predecessor, the new enumeration is one-to-one: each ruleset is guaranteed to appear exactly once in the new enumeration, which moreover possesses an elegant simplicity that allows jumps over increasingly longer undesired subsequences of many types. This process effectively results in an increased acceleration and greatly improved performance.

Keywords: sequential substitution system; enumeration

1. Introduction

Stephen Wolfram's groundbreaking enumeration of all elementary cellular automata was introduced as a showcase example in his book *A New Kind of Science* (NKS) [1]. The NKS approach is to observe the behavior of a large set of "simple programs," noticing characteristics that resemble phenomena in the natural world, perhaps ultimately providing insights about the real universe. Critical to this method is the existence of an enumeration, which for our purposes is a well-defined way of sequentially generating all possible simple programs of the desired type. Based on initial programs and visualizations developed by one of us [Caviness] during the 2009 NKS Summer School program in Pisa [2], a complete enumeration of all rulesets of sequential substitution systems (SSSs, informally pronounced "sessies") was developed [3]. Initial use of that enumeration revealed the unsurprising result that most SSSs die out (stop changing), and most of those that do not die out are of simple, repetitive types. The human researcher must be provided with interesting cases to investigate without the need to hunt through billions of cases that die out or are functionally equivalent to ones previously studied. Therefore, it became imperative that the program do preliminary testing and only display those of possible interest. Of particular importance was the existence in the enumeration of long subsequences of rulesets that could all be eliminated by the same criteria, suggesting the possibility of not merely simplifying the researcher's task but also accelerating the iteration. However, the complicated nature of the ranking/unranking algorithms caused difficulties in this effort. So while further progress on SSSs was blocked, we turned to laying the initial groundwork for the treatment of multiway and other substitution systems, developing a more generalized enumeration. Ironically, we found that not only does the new generalized substitution system (GSS) enumeration allow the treatment of cases not included in the SSS enumeration, but its simple ranking/unranking functions make possible precisely calculated jumps over subsequences of unwanted cases. This feature makes it preferable for use even with SSSs. Equally satisfactory is the fact that the new algorithm never produces the same ruleset from different indices, eliminating the 25% duplication inherent in the original SSS enumeration. Adding in non-SSS rulesets turns out not to be a disadvantage at all!

2. Substitution Systems and the Old Sequential Substitution System Enumeration

A SSS consists of a finite-length state string containing characters from an arbitrary enumerable (finite or countably infinite) alphabet and a ruleset containing one or more rules. Each rule consists of a pattern to search for, specified by the left-hand side of the rule, and a replacement string, given by the right-hand side. For example, the ruleset {"AA"→"BA", "AB"→"AA", "B"→"AA"} contains three rules, the first of which replaces occurrences of "AA" in the state string by "BA". The term sequential refers here to the prescribed substitution order: the first rule is applied only at the first possible matching location in the state string, scanning from left to right. If no match for the first rule exists, the second rule is tried in the same way, scanning the state string from left to right, and only if both the first and second rules fail is the third used, and so on [1]. In the example above, starting with an initial state string "B", applications of the rules successively produce the following state strings: {"B", "AA", "BA", "AAA", "BAA", "BBA", "AABA", "BABA", "BAAA", "BBAA", …}. (In this case, the rules invoked were {3, 1, 3, 1, 1, 3, 1, 2, 1, 1,…}.) Figure 1 is a visual representation of this SSS, showing the evolution of the state string by successive rows of colored squares (arbitrarily using gray and red to represent "A" and "B", respectively), with the rule applied and the location of each match indicated by the inset numbers. The reader can easily confirm that each row is obtained from the previous row by applying the rule at the place shown. Each SSS is associated with a well-defined causal network [1]. The SSS causal network shown in Figure 1 is the first two-dimensional network found with an internal hexagonal arrangement of nodes.

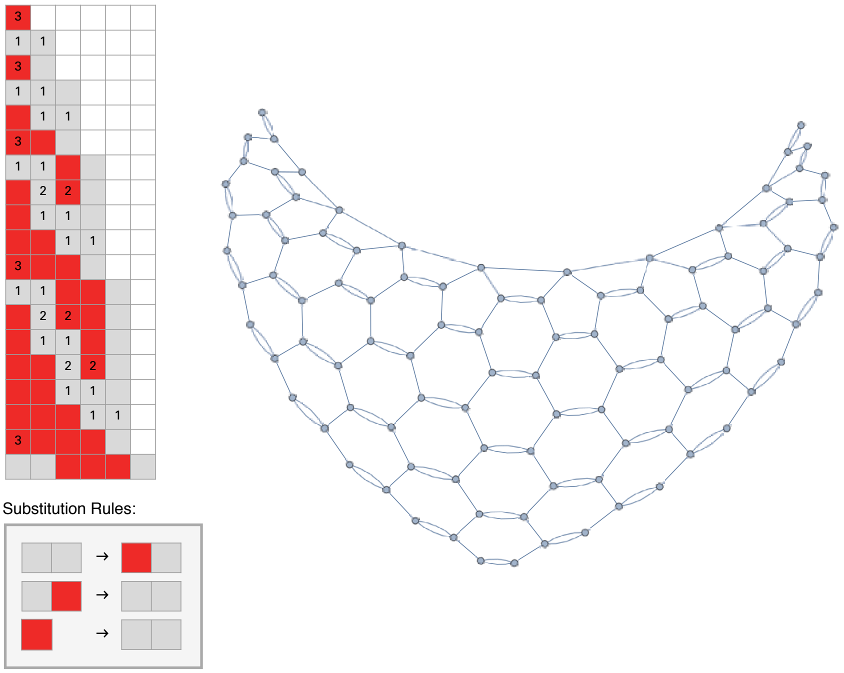

Figure 1. Visual display of ruleset, early steps of evolution and causal network of a sample SSS.

If no rules apply anywhere in the state string, no further changes can occur, and the system is said to be dead. Although we set no bound on the number of rules or string length (except that both be finite), and even the alphabet used may be countably infinite, the set of all SSS rulesets is enumerable and an enumeration was developed as detailed previously [3]. The original SSS enumeration allowed empty strings in certain positions, to accommodate deletion rules ("something"→"nothing") anywhere, and explicitly added the possibility of a creation rule ("nothing"→"something") at the end of the ruleset. (This seemed natural since no rule following a creation rule will ever be executed.) Unfortunately, the attempt to avoid unwanted cases resulted in an increase in the complexity and in the introduction of duplicate rulesets, with 25% of the index codes producing duplicates. Worse, in allowing deletion rules anywhere in the ruleset, the enumeration also allowed creation rules anywhere, rather than only at the end as was originally planned. It is true that adjacent empty strings were only allowed at the end of the ruleset, but this turns out to be a relatively small advantage, when weighed against the enumeration's complexity.

By moving to the generalized enumeration, we simplify the ranking/unranking functions and the calculation of jumps within the enumeration. We allow one or at most two adjacent empty strings anywhere in the ruleset, so adjacent deletion and creation rules are possible anywhere, for example, {"AA"→"", ""→"A"}, but now it is a simple matter to skip an entire run of unwanted nonfinal creation rules in one jump. In addition, the new enumeration can without further modification represent multiway substitution systems (which have different criteria than SSSs). The new algorithm is appropriately called the GSS enumeration, as it can be used for any such system. Its relative simplicity makes it superior to the previously used enumeration for sequential systems, and therefore the GSS enumeration can be considered a replacement for that algorithm.

3. Motivation and Construction of the Generalized Substitution System Enumeration Algorithm

The GSS enumeration is based on a quinary (base-5) code specifying how the ruleset should be constructed by repeatedly modifying a single ruleset containing a single character "A". It is a consistent, logical extension of the enumeration of all lists of all strings, which is itself an extension of the universal string enumeration, using ternary and binary codes, respectively. It is instructive to briefly review these simpler cases.

To construct any string in the universal string enumeration from its index, one starts by writing the index in binary form. The initial 1 bit is the signal to start the string with a single character "A", and any additional bits are taken as instructions to do one of the following:

- 0: append an "A" to the string
- 1: increment the final character of the string

Thus the string is lengthened by each 0 bit to the index code, while the final character is changed to the next one in the (arbitrary) alphabet by each 1 bit. For example, the bits of the binary number 101011 are interpreted as the instructions given in Table 1, resulting in the string "ABC".

Any string can be formed in this way: the process describes a bijective mapping of the positive integers onto the set of all strings. Table 2 gives the first few strings in enumeration order, together with their indices in decimal and binary form. It should be noted that any alphabet can be used for the characters actually making up the strings; here "ABC..." is used merely for convenience. It is quite conceivable to use the current complete Unicode character listing, even extending it as new characters may someday be added. Any such changes would automatically be used by our enumeration algorithm, which therefore is truly a universal list of strings, not merely of currently compiled languages but of all future human writing systems!

bit	meaning	result
1	start with "A"	"A"
0	append an "A"	"AA"
1	increment final character	"AB"
0	append an "A"	"ABA"
1	increment final character	"ABB"
1	increment final character	"ABC"

Table 1. Example of string construction from a binary code.

1	2	3	4	5	6	7	8	9	10
1_2	10_2	11_2	100_2	101_2	110_2	111_2	1000_2	1001_2	1010_2
A	AA	B	AAA	AB	BA	C	AAAA	AAB	ABA

Table 2. The beginning of the universal list of all strings of all letters, characters and symbols, together with their indices. The initial 1 in the binary code represents the original "A"; subsequent bits specify modifications ($0 \mid 1 \to$ append | increment).

Interestingly, the initial 1 bit can be thought of as adding 2^{w-1} to the binary instruction code given by subsequent digits, where w is the total "weight" of the string (formed by adding the character weights: "A"$\to$1, "B"$\to$2, "C"$\to$3, etc.) and 2^{w-1} is simply the number of strings of weight less than w that precede all the strings of weight w, plus 1. For example, as shown in Table 2, there are $2^0 = 1$ strings of weight 1, $2^1 = 2$ strings of weight 2 and $2^2 = 4$ strings of weight 3. This pattern continues: the weight of each string is the length of its binary index code, so there are 2^{w-1} strings of weight w. The number of strings of weight less than w is therefore $2^0 + 2^1 + \cdots + 2^{w-2} = 2^{w-1} - 1$, the sum of a finite geometric series. We further note that within each string weight class, larger characters appear later than equivalent weight runs of smaller characters as the weight shifts toward the beginning of the string. For example, "AAA" appears before "C". The reader is invited to experiment with online interactive demonstrations of this algorithm [4, 5].

An enumeration of all lists of strings can be obtained by again starting with a single "A", treating the indices as base-3 numbers, letting the three ternary digits encode the following instructions:

- 0: end this string and start a new string with an "A"
- 1: append an "A" to the last string
- 2: increment the last character of the last string

Then add the ternary code to the number of smaller weight string lists, plus 1, so as to start the enumeration with index 1 rather than 0. The interested reader is directed to [3] or [5] for details, but again lower-weight string lists appear before higher-weight ones, and within each weight class the weight is initially spread out (as w separate "A" strings) and is gradually compacted and shifted toward the beginning of the string.

In substitution systems, it is necessary to allow empty strings (for creation or deletion rules) and even at times two adjacent empty strings, if a deletion rule is immediately followed by a creation rule, such as in {"A"→"", ""→"A"}, for example. This motivates the use of base-5 codes for the GSS enumeration, with the following quinary digit meanings:

- 0: end this string, insert two empty strings and start a new string with an "A"
- 1: end this string, insert one empty string and start a new string with an "A"
- 2: end this string and start a new string with an "A"
- 3: end this character and start a new character (as an "A")
- 4: increment this character

One minor adjustment must be considered: there are two possible rulesets of weight 1: {""→"A"} and {"A"→""}. To guarantee that all substitution system rulesets appear in the enumeration, simply select one of these, indicating the choice by a 0 or 1 bit, before adding the base-5 digits, giving full instructions for the construction for all weight w rulesets by the $2\left(5^{w-1}\right)$ codes of the form $b_1 q_2 q_3 \ldots q_w$.
The number of rulesets of weight less than w is

$$2\left(5^0\right) + 2\left(5^1\right) + \cdots + 2\left(5^{w-2}\right) = \frac{2\left(1 - 5^{w-1}\right)}{\left(1 - 5\right)} = \frac{\left(5^w - 5\right)}{10} \tag{1}$$

and the offset to add to the quinary code to give a unique index, starting with 1, for each ruleset of weight w is

$$\frac{\left(5^w - 5\right)}{10} + 1 = \frac{\left(5^w - 5\right)}{10}. \tag{2}$$

Clearly, the useful features of the string enumeration also have their counterparts in the GSS enumeration. Each quinary digit adds one to the weight of the ruleset (defined as the sum of the weights of its strings), so the length of the code string is by construction determined by the weight of the ruleset. Using the proper offset for each weight class ensures that small-weight rulesets appear before larger-weight ones, and that each ruleset is associated with a unique index. The enumeration includes all rulesets constructible on a finite or countably infinite alphabet, and small characters and short strings appear early in the list—a significant advantage, as we will see. Figure 2 gives Wolfram Language code for the GSS rank and unrank functions, and Table 3 shows the first 75 rulesets in the GSS enumeration together with their indices. The ordering by ruleset weight (defined as the sum of the weights of the strings in the ruleset) is clearly apparent, as is the pattern of the weight being initially as spread out as possible—single character "A" strings separated by two empty strings (the maximum allowed) and gradually being compacted and moved toward the beginning of the ruleset. Still, many rulesets appear that are not needed for SSSs, notably cases of nonfinal creation rules (and even ""→"" rules), but as we will show in Section 5, the underlying simplicity and coherence of enumeration allow such cases to be jumped over with little loss of time. The most important feature is that the enumeration does not leave anything out, and any "extras" can be managed efficiently.

1	$0_2\,{}_5$	{"" → "A"}
2	$1_2\,{}_5$	{"A" → ""}
3	$0_2 0_5$	{"" → "A", "" → "", "A" → ""}
4	$0_2 1_5$	{"" → "A", "" → "A"}
5	$0_2 2_5$	{"" → "A", "A" → ""}
6	$0_2 3_5$	{"" → "AA"}
7	$0_2 4_5$	{"" → "B"}
8	$1_2 0_5$	{"A" → "", "" → "A"}
9	$1_2 1_5$	{"A" → "", "A" → ""}
10	$1_2 2_5$	{"A" → "A"}
11	$1_2 3_5$	{"AA" → ""}
12	$1_2 4_5$	{"B" → ""}
13	$0_2 00_5$	{"" → "A", "" → "", "A" → "", "" → "A"}
14	$0_2 01_5$	{"" → "A", "" → "", "A" → "", "A" → ""}
15	$0_2 02_5$	{"" → "A", "" → "", "A" → "A"}
16	$0_2 03_5$	{"" → "A", "" → "", "AA" → ""}
17	$0_2 04_5$	{"" → "A", "" → "", "B" → ""}

Table 3. (*continues*)

18	$0_2 10_5$	{"" → "A", "" → "A", "" → "", "A" → ""}
19	$0_2 11_5$	{"" → "A", "" → "A", "" → "A"}
20	$0_2 12_5$	{"" → "A", "" → "A", "A" → ""}
21	$0_2 13_5$	{"" → "A", "" → "AA"}
22	$0_2 14_5$	{"" → "A", "" → "B"}
23	$0_2 20_5$	{"" → "A", "A" → "", "" → "A"}
24	$0_2 21_5$	{"" → "A", "A" → "", "A" → ""}
25	$0_2 22_5$	{"" → "A", "A" → "A"}
26	$0_2 23_5$	{"" → "A", "AA" → ""}
27	$0_2 24_5$	{"" → "A", "B" → ""}
28	$0_2 30_5$	{"" → "AA", "" → "", "A" → ""}
29	$0_2 31_5$	{"" → "AA", "" → "A"}
30	$0_2 32_5$	{"" → "AA", "A" → ""}
31	$0_2 33_5$	{"" → "AAA"}
32	$0_2 34_5$	{"" → "AB"}
33	$0_2 40_5$	{"" → "B", "" → "", "A" → ""}
34	$0_2 41_5$	{"" → "B", "" → "A"}
35	$0_2 42_5$	{"" → "B", "A" → ""}
36	$0_2 43_5$	{"" → "BA"}
37	$0_2 44_5$	{"" → "C"}
38	$1_2 00_5$	{"A" → "", "" → "A", "" → "", "A" → ""}
39	$1_2 01_5$	{"A" → "", "" → "A", "" → "A"}
40	$1_2 02_5$	{"A" → "", "" → "A", "A" → ""}
41	$1_2 03_5$	{"A" → "", "" → "AA"}
42	$1_2 04_5$	{"A" → "", "" → "B"}
43	$1_2 10_5$	{"A" → "", "A" → "", "" → "A"}
44	$1_2 11_5$	{"A" → "", "A" → "", "A" → ""}
45	$1_2 12_5$	{"A" → "", "A" → "A"}
46	$1_2 13_5$	{"A" → "", "AA" → ""}
47	$1_2 14_5$	{"A" → "", "B" → ""}
48	$1_2 20_5$	{"A" → "A", "" → "", "A" → ""}
49	$1_2 21_5$	{"A" → "A", "" → "A"}
50	$1_2 22_5$	{"A" → "A", "A" → ""}
51	$1_2 23_5$	{"A" → "AA"}
52	$1_2 24_5$	{"A" → "B"}
53	$1_2 30_5$	{"AA" → "", "" → "A"}
54	$1_2 31_5$	{"AA" → "", "A" → ""}

Table 3. (*continues*)

56	$1_2 33_5$	{"AAA" → ""}
57	$1_2 34_5$	{"AB" → ""}
58	$1_2 40_5$	{"B" → "", "" → "A"}
59	$1_2 41_5$	{"B" → "", "A" → ""}
60	$1_2 42_5$	{"B" → "A"}
61	$1_2 43_5$	{"BA" → ""}
62	$1_2 44_5$	{"C" → ""}
63	$0_2 000_5$	{"" → "A", "" → "", "A" → "", "" → "A", "" → "", "A" → ""}
64	$0_2 001_5$	{"" → "A", "" → "", "A" → "", "" → "A", "" → "A"}
65	$0_2 002_5$	{"" → "A", "" → "", "A" → "", "" → "A", "A" → ""}
66	$0_2 003_5$	{"" → "A", "" → "", "A" → "", "" → "AA"}

Table 3. The beginning of the universal list of all substitution system rulesets, using the GSS enumeration.

```
fromGeneralizedRank[i_Integer /; i > 0] := Module[
        {n, j, bit, quinaryDigits, numberOfEOS, chopPos, ans, strings, ruleset},
        n = Floor[Log[5, 10 i - 5]];
        j = i - (5^n + 5) / 10;
        {bit, j} = QuotientRemainder[j, 5^(n-1)];
        quinaryDigits = IntegerDigits[j, 5, n - 1];
        ans = Switch[bit, 0, {{}, {1}}, 1, {{1}}];
        Scan[Switch[#,
                0 , ans = Join[ans, {{}, {}, {1}}] ,
                1, ans = Join[ans, {{}, {1}}] ,
                2, AppendTo[ans, {1}],
                3, AppendTo[ans[[-1]], 1],
                4, ans[[-1]][[-1]] ++
            ] &, quinaryDigits];
        strings = StringJoin @@@ (FromCharacterWeights /@ ans);
        If[OddQ[Length[strings]], strings = AppendTo[strings, ""]];
        Rule @@@ Partition[strings, 2, 2]];

toGeneralizedRank[rs_List] := Module[{rl, wl, w, bit, quincode = ""},
      rl = Flatten[List @@@ rs];
      If[Last[rl] == "", rl = Most[rl]]; (* drop ultimate empty string, if needed *)
      wl = ToCharacterWeights /@ rl;
      (* to lists of lists of numbers "A"→1, etc., but ""→{}, not 0 *)
      wl = wl /. 0 → {};
      w = Total[Flatten[wl]]; (* weight of this rule set *)
```

Figure 2. (*continues*)

```
While[wl ≠ {{1}} && wl ≠ {{}, {1}},
        Which[
          wl⟦-1⟧⟦-1⟧ > 1, quincode = "4" <> quincode; wl⟦-1⟧⟦-1⟧--,
          wl⟦-1⟧⟦-1⟧ == 1 && Length[wl⟦-1⟧] > 1,
          quincode = "3" <> quincode; wl⟦-1⟧ = Most[wl⟦-1⟧],
Length[wl] ≥ 3 && wl⟦-3 ;;⟧ == {{}, {}, {1}}, quincode = "0" <> quincode;
          wl = Drop[wl, -3],
          Length[wl] ≥ 2 && wl⟦-2 ;;⟧ == {{}, {1}},
          quincode = "1" <> quincode; wl = Drop[wl, -2],
          wl⟦-1⟧ == {1}, quincode = "2" <> quincode; wl = Drop[wl, -1]
        ]];
    Switch[wl, {{}, {1}}, bit = 0, {{1}}, bit = 1];
    FromDigits[bit <> quincode, 5] + (5^w + 5) / 10];
```

Figure 2. The GSS rank and unrank functions.

4. The Reduced Enumeration Substitution System Algorithm

The GSS enumeration includes all rulesets for general substitution systems. It can be used equally well for the treatment of SSSs, particularly when implemented with long jumps to bypass runs of consecutive rulesets that can be omitted in the study of SSSs. Several long-jump criteria for SSSs will be considered in Section 5. Of course, the same can be done for other types of substitution systems as well, although the criteria will differ. But one simplification worth considering first is removing the initial binary code used to distinguish between rulesets starting with a creation rule (""→"something") and those that do not. Particularly for the SSS enumeration, where the only rulesets needing consideration are those without creation rules or with one creation rule as the final rule in the ruleset, eliminating in advance all initial creation rules could be advantageous, halving the number of cases to consider. This can be done almost trivially by dropping the initial bit of the quinary code and using the instructions to operate on the initial ruleset {"A"→""} in all cases. The only additional modification needed is to change the offset for weight class w rulesets to $\left(5^{w-1}+3\right)/4$, calculated as one more than the number of rulesets of weight less than w.

The reduced enumeration substitution system (RSS) algorithm, which includes all SSS cases we might be interested in, with the exception of rulesets containing a single creation rule, {""→"something"}, is shown in Figure 3, and the beginning of the list generated is shown in Table 4. Long jumps over nonfinal creation rules will still be needed, in addition to skipping over runs of omittable rulesets of other types. It should be noted, however, that this strategy will also remove singleton creation rules such as {""→"A"}, which would not be excluded from the SSS case by other considerations. But in fact all such SSS cases very closely resemble each other and can be treated separately if desired.

1	$_5$	{"A" → ""}
2	0_5	{"A" → "", "" → "A"}
3	1_5	{"A" → "", "A" → ""}
4	2_5	{"A" → "A"}
5	3_5	{"AA" → ""}
6	4_5	{"B" → ""}
7	00_5	{"A" → "", "" → "A", "" → "", "A" → ""}
8	01_5	{"A" → "", "" → "A", "" → "A"}
9	02_5	{"A" → "", "" → "A", "A" → ""}
10	03_5	{"A" → "", "" → "AA"}
11	04_5	{"A" → "", "" → "B"}
12	10_5	{"A" → "", "A" → "", "" → "A"}
13	11_5	{"A" → "", "A" → "", "A" → ""}
14	12_5	{"A" → "", "A" → "A"}
15	13_5	{"A" → "", "AA" → ""}
16	14_5	{"A" → "", "B" → ""}
17	20_5	{"A" → "A", "" → "", "A" → ""}
18	21_5	{"A" → "A", "" → "A"}
19	22_5	{"A" → "A", "A" → ""}
20	23_5	{"A" → "AA"}
21	24_5	{"A" → "B"}
22	30_5	{"AA" → "", "" → "A"}
23	31_5	{"AA" → "", "A" → ""}
24	32_5	{"AA" → "A"}
25	33_5	{"AAA" → ""}
26	34_5	{"AB" → ""}
27	40_5	{"B" → "", "" → "A"}
28	41_5	{"B" → "", "A" → ""}
29	42_5	{"B" → "A"}
30	43_5	{"BA" → ""}
31	44_5	{"C" → ""}
32	000_5	{"A" → "", "" → "A", "" → "", "A" → "", "" → "A"}
33	001_5	{"A" → "", "" → "A", "" → "", "A" → "", "A" → ""}
34	002_5	{"A" → "", "" → "A", "" → "", "A" → "A"}
35	003_5	{"A" → "", "" → "A", "" → "", "AA" → ""}

Table 4. (*continues*)

36	004_5	{"A" → "", "" → "A", "" → "", "B" → ""}
37	010_5	{"A" → "", "" → "A", "" → "A", "" → "", "A" → ""}
38	011_5	{"A" → "", "" → "A", "" → "A", "" → "A"}
39	012_5	{"A" → "", "" → "A", "" → "A", "A" → ""}
40	013_5	{"A" → "", "" → "A", "" → "AA"}
41	014_5	{"A" → "", "" → "A", "" → "B"}
42	020_5	{"A" → "", "" → "A", "A" → "", "" → "A"}
43	021_5	{"A" → "", "" → "A", "A" → "", "A" → ""}
44	022_5	{"A" → "", "" → "A", "A" → "A"}
45	023_5	{"A" → "", "" → "A", "AA" → ""}
46	024_5	{"A" → "", "" → "A", "B" → ""}
47	030_5	{"A" → "", "" → "AA", "" → "", "A" → ""}
48	031_5	{"A" → "", "" → "AA", "" → "A"}
49	032_5	{"A" → "", "" → "AA", "A" → ""}
50	033_5	{"A" → "", "" → "AAA"}
51	034_5	{"A" → "", "" → "AB"}
52	040_5	{"A" → "", "" → "B", "" → "", "A" → ""}
53	041_5	{"A" → "", "" → "B", "" → "A"}
54	042_5	{"A" → "", "" → "B", "A" → ""}
55	043_5	{"A" → "", "" → "BA"}
56	044_5	{"A" → "", "" → "C"}
57	100_5	{"A" → "", "A" → "", "" → "A", "" → "", "A" → ""}
58	101_5	{"A" → "", "A" → "", "" → "A", "" → "A"}
59	102_5	{"A" → "", "A" → "", "" → "A", "A" → ""}
60	103_5	{"A" → "", "A" → "", "" → "AA"}
61	104_5	{"A" → "", "A" → "", "" → "B"}
62	110_5	{"A" → "", "A" → "", "A" → "", "" → "A"}
63	111_5	{"A" → "", "A" → "", "A" → "", "A" → ""}
64	112_5	{"A" → "", "A" → "", "A" → "A"}
65	113_5	{"A" → "", "A" → "", "AA" → ""}
66	114_5	{"A" → "", "A" → "", "B" → ""}

Table 4. The beginning of the RSS enumeration.

```
fromReducedRank[i_Integer /; i > 0] := Module[{n, j, quinaryDigits,
      numberOfEOS, chopPos, extra, ans = {{1}}, strings, ruleset},
   n = Floor[Log[5, 4 i - 3]];
   j = i - (5^n + 3) / 4;
   quinaryDigits = IntegerDigits[j, 5, n];
   (* the base-5 code for this ruleset will contain n digits,
   the ruleset weight is n+1 *)
   Scan[Switch[#,
         0 , ans = Join[ans, {{}, {}, {1}}] ,
         1, ans = Join[ans, {{}, {1}}] ,
         2, AppendTo[ans, {1}],
         3, AppendTo[ans[[-1]], 1],
         4, ans[[-1]][[-1]] ++
      ] &, quinaryDigits];
   strings = StringJoin @@@ (FromCharacterWeights /@ ans);
   If[OddQ[Length[strings]], strings = AppendTo[strings, ""]];
   Rule @@@ Partition[strings, 2, 2]];
toReducedRank[rs_List] := Module[{rl, wl, w, code = ""},
   rl = Flatten[List @@@ rs];
   If[Last[rl] == "", rl = Most[rl]];
   (* drop ultimate empty string, if needed *)
   wl = ToCharacterWeights /@ rl; (* to lists of lists of numbers "A"→1,
   etc., but ""→{}, not 0 *)
   wl = wl /. 0 → {};
   w = Total[Flatten[wl]]; (* weight of this rule set *)
   While[wl ≠ {{1}},
     Which[
       wl[[-1]][[-1]] > 1, code = "4" <> code; wl[[-1]][[-1]] --,
       wl[[-1]][[-1]] == 1 && Length[wl[[-1]]] > 1,
       code = "3" <> code; wl[[-1]] = Most[wl[[-1]]],
       Length[wl] ≥ 3 && wl[[-3 ;;]] == {{}, {}, {1}},
       code = "0" <> code; wl = Drop[wl, -3],
       Length[wl] ≥ 2 && wl[[-2 ;;]] == {{}, {1}},
       code = "1" <> code; wl = Drop[wl, -2],
       wl[[-1]] == {1}, code = "2" <> code; wl = Drop[wl, -1]
     ]];
   (FromDigits[code, 5] + (5^(w-1) + 3) / 4)
   (* add number of smaller weight rulesets *) ];
```

Figure 3. The RSS rank and unrank functions.

5. Criteria for Skipping Sequential Substitution System Rulesets and Long-Jump Calculations

Even the RSS enumeration includes many rulesets that we can discard immediately. For example, {"A"→"", ""→"A", ""→"", "A"→""} (#7 in the RSS list, quinary code 005) would be eliminated for several reasons: it includes an identity rule (where a string is replaced by itself) and several pairs of conflicting rules (e.g., where different rules have the same left-hand side, thereby giving conflicting instructions). Such problems, as explained in detail in the following, guarantee that the SSS generated by this ruleset will be identical to that of a simpler one that has already appeared earlier in the enumeration. Omitting such cases before any more time is invested greatly speeds up any treatment of SSSs. Even more significantly, due to the nature of the enumeration algorithm, cases that would be skipped for the same reason occur consecutively, allowing us to jump past all similar cases without any testing at all. In this section we consider four of the most productive criteria for these "long jumps" and how the jump size can be calculated.

5.1 Conflicting Rules

A pair of conflicting rules occurs when two rules in the ruleset have the same left-hand side, or more generally, when the left-hand side of one rule is a substring of the left-hand side of a later rule. For example, consider the ruleset {"ABA"→"AAB", "ABA"→"", "A"→"ABA"} (# 158 448 842 380 in the RSS list, quinary code = $3\,432\,334\,234\,312\,343_5$).

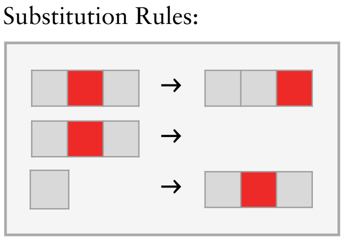

Figure 4. Visualization of the ruleset {"ABA"→"AAB", "ABA"→"", "A"→"ABA"}.

A conflicting rule situation ensures that some rule in the set will never be used. In this case, whenever there is an "ABA" to be found, the first rule will be executed. The second rule will never be invoked, since it is only considered if the first rule fails. Therefore, the ruleset is functionally equivalent to a simpler ruleset that appears earlier in the enumeration: {"ABA"→"AAB", "A"→"ABA"} (# 253 518 005 in the RSS list, quinary code = $343\,233\,422\,343_5$).

Every conflicting rules case identified is actually one of a run of rulesets in the enumeration having the same problem. Rather than testing each of these rulesets separately, to accelerate the enumeration we can jump directly to the last ruleset in the run of those sharing the particular problem, or to the first that does not (i.e., the resolution of the problem), either using knowledge of the enumeration or by performing an operation on the quinary code.

It is helpful to consider the first ruleset in the run of rulesets sharing the problem. For this, we need to look at where the conflicting rule is located in the ruleset. A shortcut to finding the quinary code digit that caused the conflict is to add up the combined weight of the remaining characters in the ruleset after the problem. In the example above, the conflicting (second) rule is "ABA"→"", but the actual problem is due to the left-hand side of this rule, "ABA". The remaining weight after the problem is 5, found by adding the weights of all subsequent strings in the ruleset: "", "A", "ABA" (0 + 1 + 1 + 2 + 1). The quinary code for our example above is $3\,432\,334\,234\,312\,343_5$, and changing the rightmost five digits to 0 will result in a quinary code of the first problem ruleset in the run, the first one to start with {"ABA"→"AAB", "ABA"→...}, followed by some distribution of strings having weight 5. Here the first problem ruleset has the quinary code of $3\,432\,334\,234\,300\,000_5$, which translates to {"ABA"→"AAB", "ABA"→"", ""→"A ", ""→"", "A "→"", ""→"A ", ""→"", "A "→"", ""→"A "} (# 158 448 841 407 in the RSS list). Operating directly on the quinary code, we have truncated the final five digits and then appended five 0s. Mathematically this can be accomplished for remaining weight w by dividing the base-5 number by 5^w, discarding any remainder, then multiplying by 5^w, effectively replacing the last w digits by 0s. Reminder: to generate the quinary code, subtract $\left(5^{w-1}+3\right)/4$ from the actual RSS index and then write the result as a base-5 number, where w is the weight of the ruleset, found by adding up the weights of all the characters in all the strings, or from the index i: $w = \text{IntegerPart}\left[\log_5(4i-3)\right]+1$.

Once the first conflicting ruleset that shares the problem is known, the first ruleset that does not share the problem can be easily found, again by replacing the last w digits: the first by a 4 and the rest by 0s. Mathematically, this means adding $4\left(5^{w-1}\right)$ to the RSS enumeration index of the first problem that occurs. In this example, the first ruleset without the problem would be {"ABA"→"AAB", "ABB"→"", ""→"A", ""→"", "A "→"", ""→"A ", ""→"", "A "→""} (# 158 448 843 907 in the RSS list, quinary code $3\,432\,334\,234\,340\,000_5$). The reason this resolution ruleset no longer contains the problematic conflicting rule is because the 4 incremented the last character of the string being constructed, changing the second "ABA" string to "ABB". Appending 0s to the quinary code completes the desired weight, giving the first ruleset after the run of problematic cases.

One exception to this process is when the conflicting rules situation involves an empty string, that is, a creation rule. An empty string in the left-hand side of any rule except the last one will automatically conflict with any rule that follows it. The only way to avoid this situation is to have no further rules, achieved by shifting the weight of any subsequent rules into the right-hand side of the (now final) creation rule. There are two possible cases: (0) If the creation rule was created by a 0 in the quinary code, in the resolution code the 0 becomes a 1 and all later digits are changed to 3, effectively making the ""→ "" + weight-w rule sequence into ""→"AAA…A", with w "A" characters. (1) If the creation rule was created by a 1, the right-hand side is not empty (and not a problem), but any additional rules' weight must be appended to it as a string of "A" characters. Find the combined weight w of any rules following the creation rule (past its right-hand side), and replace the last w quinary code digits by 3s.

	Ruleset	Quinary Code
Problem	{"ABA" → "AAB", "ABA" → "", "A" → "ABA"}	3 432 334 234312 343$_5$
Resolution	{"ABA" → "AAB", " AB B " → "", …}	3 432 334 234340 000$_5$
Problem	{"A" → "BC", "AB" → "C"}	24 344234 244$_5$
Resolution	{"A" → "BC", "B" → "", …}	24 344240 000$_5$
Problem	{"BC" → "A", "BC" → "B", "A" → "B"}	434 422 434424 224$_5$
Resolution	{"BC" → "A", " B D " → "", …}	434 422 434440 000$_5$
Problem	{"A" → "B", "" → "C", "" → "A", "B" → "AB"}	24144124 234$_5$
Resolution	{"A" → "B", "" → " C AAAAAA "}	24144333 333$_5$
Problem	{"A" → "B", "" → "CB", "" → "A", "B" → "AB"}	2414 434124 234$_5$
Resolution	{"A" → "B", "" → " CB AAAAAA "}	2414 434333 333$_5$
Problem	{"A" → "B", "" → "", "B" → "AB"}	2404234$_5$
Resolution	{"A" → "B", "" → "AAAAA"}	2413 333$_5$

Table 5. Conflicting rule examples of two types.

It may be noted that the normal resolution process of the first type described above (replacing the final w quinary digits by 4 followed by 0s) usually creates a new conflicting rules situation of the second type, which, in turn, will be resolved in the next step.

5.2 Identity Rule

An identity rule occurs when the left-hand side of a rule is identical to the right-hand side of that same rule. A ruleset containing an identity rule and at least one other rule can be discarded, whether or not the identity rule is ever executed. For example, consider the ruleset {"A"→ "CC", "B"→"B", "C"→"A"} (# 5 178 059 904 in RSS list, quinary code = 24 434 424 242 442$_5$). If the second rule is never executed, the behavior of the ruleset is equivalent to a simpler, previously considered one, without the identity rule: {"A"→"CC", "C"→"A"} (# 8 284 904, quinary code = 2 443 442 442$_5$). On the other hand, if the identity rule is executed once, the enumeration will go into an infinite loop of "B"→"B" and any later rules will never be executed afterwards.

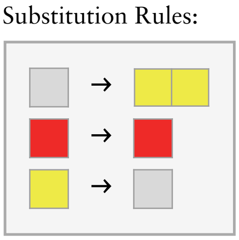

Figure 5. Visualization of the ruleset {"A"→"CC", "B"→"B", "C"→"A"}.

Again, this ruleset is part of a subsequence in the enumeration, all starting the same way and including the same identity rule. To be able to jump over them all, we must first identify where in the ruleset the problem occurs. This is done by looking at the left-hand and right-hand side of each rule and checking whether they are identical. Then we identify which quinary code digit caused the problem and zero out the quinary code after the problem digit to form the first ruleset with this problem. In the example above, the identity rule is followed by the strings "C" and "A", having combined weight 3 + 1 = 4, so it was the fifth digit from the last in the quinary code $24\,434\,424\,242\,442_5$ that created the "B" in the right-hand side of the identity rule "B"→"B". Changing the last four digits to 0 gives $24\,434\,424\,240\,000_5$, which corresponds to the ruleset {"A"→"CC", "B"→"B", ""→"", "A"→"", ""→"A", ""→"", "A"→""} (# 5 178 059 532 in the RSS list), the first in our run sharing the same initial rules up to and including this particular identity rule problem.

The process of finding the resolution follows a similar method. Once the identity rule is found and identified, change the digit of the quinary code directly after the one that caused the identity problem to a 3 (an append instruction, which changes the current string by adding an "A" to it) and then zero out the remaining digits. In our example, we would change the quinary code $24\,434\,424\,242\,442_5$ to $24\,434\,424\,243\,000_5$, which corresponds to the ruleset {"A"→ "CC", "B"→"BA", ""→"", "A"→"", ""→"A", ""→"", "A"→""} (# 1 027 669 282 in the RSS list). Again, starting with quinary code $2\,434\,424\,240\,000_5$, all rulesets in the enumeration, up to but not including the resolution $2\,434\,424\,243\,000_5$, share the same first rule and the identity rule, a situation that allows a long jump over this entire subsequence of the enumeration.

The mathematical algorithm for this process is: truncate the last w digits of the quinary code (dividing by 5^w and taking the integer part), then multiply by 5^w, and finally add $3\left(5^{w-1}\right)$, where w is the weight of the strings after the identity rule.

The exception to this algorithm is when the identity rule involves two empty strings: finding the resolution in this case involves a process slightly different from the one described above. The quinary digit that created a ""→"" identity rule is necessarily a 0, and the first ruleset in the problem run can still be found by zeroing out the remaining quinary digits. But to find the resolution, we change the problem digit itself to a 1 (an instruction that creates only one empty string and then starts a new string with an "A" without creating the problematic second empty string), and then fill the remaining digits by 0s. Mathematically, if w is the weight of the strings after a ""→"" identity rule, we truncate the last w digits, multiply by 5^w, and finally add $1\left(5^{w-1}\right)$.

Note, however that a nonfinal ""→"" identity rule, such as shown in the last example in Table 6, is also a conflicting rules situation, whose resolution takes us farther in the RSS enumeration than the identity rule resolution shown. Here the conflicting rules resolution occurs at quinary code $2\,413\,333_5$ {"A"→"B", ""→"AAAAA"}.

	Ruleset	Quinary Code
Problem	{"C" → "A", "A" → "A", "B" → "BC"}	$442222\,424\,344_5$
Resolution	{"C" → "A", "A" → " AA ", ...}	$442223\,000\,000_5$
Problem	{"B" → "AB", "ABA" → "ABA", "A" → "BA"}	$42\,342\,343\,23432243_5$
Resolution	{"B" → "AB", " ABA " → "ABAA", ...}	$42\,342\,343\,23433000_5$
Problem	{"BC" → "BC", "A" → "B", "B" → "AC"}	$43\,442\,4344224\,242\,344_5$
Resolution	{"BC" → "BCA", ...}	$43\,442\,4344300\,000\,000_5$
Problem	{"A" → "B", "" → "", "B" → "AB"}	2404234_5
Resolution	{"A" → "B", "" → "A", ...}	$2410\,000_5$

Table 6. Identity rule examples of two types.

5.3 Initial Substring Rule

Substring rules occur when the left-hand side of a rule contains a substring of the right-hand side of the same rule. Our concern here is when the substring rule is the first in the ruleset and is not the only rule in the ruleset. For example, consider the ruleset {"A"→"BAC", "B"→"C", "C"→"B"} (# 128 230 799 521 in the RSS list, quinary code = $2\,433\,442\,424\,424\,424_5$). If the first rule is ever executed, it will go into an infinite loop, exhibiting behavior equivalent to the simpler ruleset {"A"→"BAC"}: rules 2 and 3 will never be executed again. Each invocation of rule 1 replaces the first "A" in the state string by "BAC": "A", "BAC", "BBACC", "BBBACCC" and so on.

On the other hand, if the state string has no "A", rule 1 is never invoked, and the behavior depends only on rules 2 and 3. A simple example of this situation starts with the initial string "B": "B", "C", "B", "C", ….

Which behavior occurs depends on the initial state string. But in any case, a substring rule in the first position in the ruleset reduces to one of two cases considered earlier in the enumeration, and so is effectively a duplication that can be discarded. To jump over the whole run of such cases, we first identify where the problem occurs in the ruleset. In this example, the problem occurs at the character "A" on the right-hand side of the first rule ({"A"→"BAC", "B"→"C", "C"→"B"}). The quinary code that triggered the problem is the third digit, highlighted for easy reference: $243\,3442\,424\,424\,424_5$.

To find the first ruleset where this particular initial substring rule occurs, we zero out the remaining digits after the highlighted problem digit. In this example, the quinary code of $2\,433\,442\,424\,424\,424_5$ is changed to $2\,430\,000\,000\,000\,000_5$, which corresponds to the ruleset {"A"→"BA", ""→"", "A"→"", ""→"A", ""→"", "A"→"", ""→"A", ""→"", "A"→"", ""→"A", ""→"", "A"→"", ""→"A", ""→"", "A"→"", ""→"A", ""→"", "A"→"", ""→"A", ""→"", "A"→""}. The number of digits to zero out can be found from the weight of the post-problem (sub)string(s) in the ruleset (here: "C","B","C","C","B": 3 + 2 + 3 + 3 + 2 = 13).

To skip over to the first ruleset without this conflict, we change the next quinary code digit to a 4 and zero out all digits after it. In this case, the new quinary code will be $2\,434\,000\,000\,000\,000_5$. This gives us the new ruleset of {"A"→"BB", ""→"", "A"→"", ""→"A", ""→"", "A"→"", ""→"A", ""→"", "A"→"", ""→"A", ""→"", "A"→"", ""→"A", ""→"", "A"→"", ""→"A", ""→"", "A"→"", ""→"A"} (# 128 234 863 282 in the RSS list). While this is similar to the conflicting rules situation, the quinary patterns are different enough to consider it separately. Note particularly that in the reduced SSS enumeration, the first rule of the ruleset cannot start with an empty string, eliminating a large class of initial substring rule cases and obviating the need for separate treatment of them. Note, however, that these cases are included in the GSS enumeration: the last example in Table 7 is of this type. Resolution is achieved by replacing the initial problem (binary digit 0) by a 1 and zeroing out the remaining quinary digits.

Note: a substring rule not in the initial position of the ruleset does not necessarily reduce to a simpler case. Example: {"AC"→"", "AB"→"AA", "A"→"BAC", "B"→"AA"} (#19 928 943 753 645, quinary code: $3\,441\,342\,322\,433\,442\,423_5$). Rule 3 is a substring rule, but its execution does not cause an infinite loop, since rule 1, which has precedence, can also “use up” the character "A", rather than allowing rule 3 to repeat indefinitely. Starting with an initial state string of "A" would allow all four rules to be executed according to a complicated, nonrepeating pattern, yielding, successively, the state strings {"A", "BAC", "B", "AA", "BACA", "BA", "BBAC", "BB", "AAB", "AAA", "BACAA", "BAA", "BBACA", "BBA", "BBBAC", "BBB", "AABB", "AAAB", "AAAA", …}. Unlike initial substring rule cases, rulesets with non-initial substring rules should be tested individually for possible interesting behavior.

	Ruleset	Quinary Code
Problem	{"A" → "BAC", "B" → "C", "C" → "B"}	$2433\,442\,424\,424\,424_5$
Resolution	{"A" → "BB", …}	$2434\,000\,000\,000\,000_5$
Problem	{"AB" → "ABC", "C" → "A"}	$342343\,442\,442_5$
Resolution	{"AB" → " AC ", …}	$342344\,000\,000_5$
Problem	{"" → "B", "B" → "AB"}	N/A (GSS : $0\,424\,234_5$)
Resolution	{"A" → "", …}	N/A (GSS : $1\,000\,000_5$)

Table 7. Initial substring rule examples.

5.4 Renamed Ruleset

If the characters used in the strings of the ruleset are permuted or replaced by other characters, the SSS will look the same, up to a permutation of colors, and the causal network will be identical. For example, the SSSs and causal networks generated by {"B"→"A", "A"→"B"}, {"A"→"C", "C"→"A"},…, {“I"→"U", "U"→"I"},… are all visually indistinguishable from those of {"A"→"B", "B"→"A"}, except for choice of colors to represent the different characters. For simplicity, we take as the canonical form for all such “duplicates” the first of them to appear in the enumeration (the one having the smallest RSS index), and any ruleset not in canonical form can be discarded. Verifying whether a ruleset is in canonical form can be done directly, without generating the SSS, as follows:

Examine the characters of all the strings of the ruleset in order, one character at a time. The first character must be "A" (else renaming could make it so, resulting in a lower-index ruleset), the first character that is not an "A" must be a "B", and the first that is not "A" or "B" must be "C", and so forth. As soon as an offending (nonconsecutive) character is found, take the total weight of all further characters and replace that number of quinary code digits by a 4. This gives the quinary code of the last ruleset having a too-large character at the problem position and the same pre-problem (sub)strings, so this is the last ruleset that can safely be discarded in a long jump. (The resolution ruleset is the next in the enumeration, found by incrementing the RSS index by 1.)

In Table 8, the first example is already in canonical form, since the first character is "A", the first non-"A" is a "B", and there are no non-"A" or "B" characters to consider. In the second example, the initial "B" is the problem character, the remaining weight after the problem character is 4 (1 + 1 + 2), so the quinary code of the last problematic ruleset having this weight and a problematic character in the first position ({"F"→""}) is found by replacing the last four digits by 4. The third example starts out well with an "A", but the second character is "D" rather than the canonical "B". Replacing the last five digits (4 + 1) by 4 gives the quinary code of the pre-resolution ruleset, {"A"→"I"}. In the fourth example, the post-problem weight is also five (1 + 2 + 2 + 0), again caused by having a "D" without a preceding "C". In the last example, all characters were allowed until the "E", and there is only one digit replaced by a 4.

	Ruleset	Quinary Code
Problem	{"A" → "B", "B" → "A"}	$24\,242_5$
Pre-resolution	N/A (already in canonical form)	N/A
Problem	{"B" → "A", "A" → "B"}	42224_5
Pre-resolution	{"F" → ""}	44444_5
Problem	{"A" → "D", "D" → "A"}	$244424\,442_5$
Pre-resolution	{"A" → "I"}	$244444\,444_5$
Problem	{"AB" → "D", "A" → "B", "B" → ""}	$34\,244422\,424_5$
Pre-resolution	{"AB" → "I"}	$34\,244444\,444_5$
Problem	{"A" → "ABA", "C" → "ABEA"}	$23\,432\,442\,343\,44443_5$
Pre-resolution	{"A " → "ABA", "C" → "ABF"}	$23\,432\,442\,343\,44444_5$

Table 8. Renamed ruleset examples.

In all these cases, the resolution ruleset is the next one in the enumeration, but it is more convenient to add 1 to the RSS index rather than operate on the quinary code.

Important note: Although the renamed ruleset may actually have a greater weight than the original version, this jump is safe and will allow us to skip all but one version of each ruleset (up to a permutation of the characters themselves). For example, {"A"→"BB"} (# 526, quinary code = 2434, weight: 1 + 2 + 2 = 5) is canonical but comes later in the enumeration than {"B"→"AA"} (# 145, quinary code = 423, weight: 2 + 1 + 1 = 4). It is true that we may have to wait until later in the enumeration to find some interesting case, since the renaming process preserves ruleset length, not ruleset weight. But the advantage of being able to jump over huge intervals is too good to be missed, and we can always later find the lowest-weight representative of any interesting case if desired.

6. Performance Comparisons

Implementing any one of the long-jump criteria considered above results in significant increase in the speed of the iteration, in some cases spectacular acceleration. The following table rates the computational savings for these tests, listing what fraction of the enumeration still had to be performed: the ratio of the number of cases still requiring treatment to the enumeration index n. Small numbers are better here, indicating a larger fraction of the cases skipped over. We see that long jumps triggered by conflicting rules cases are particularly effective at accelerating the enumeration.

n	Conflicting	Identity	Substring	Renamed
10^3	0.349	0.655	0.881	0.827
10^6	0.036	0.376	0.910	0.779

Table 9. Performance comparison for four long-jump tests.

As one of us (Case) repeatedly pointed out, almost all long jumps of all types are followed by a long sequence of conflicting rules cases of the second type, with conflicts due to empty strings at various places in the ruleset. Table 10 shows the great increase in performance obtained by implementing a conflicting rules check and long jump (of the creation rule type) after each long jump of other types, making the iteration loop far faster than the default $O(n)$ for three of the cases investigated. This is not unexpected, since the average jump size increases as the iteration advances to larger quinary code lengths. The last column implements all four tests together, always making the longest jump possible.

n	Conflicting	Identity	Substring	Renamed	All
10^3	0.289	0.349	0.872	0.423	$0.153 \approx 10^{-1}$
10^6	0.0286	0.0383	0.902	0.0866	$0.00971 \approx 10^{-2}$
10^9	0.00244	0.00343	0.732	0.0336	$0.000641 \approx 10^{-3}$
10^{12}	N/A	N/A	N/A	N/A	$0.000110 \approx 10^{-4}$

Table 10. Performance comparison, including an additional second type of conflicting rules jump.

One surprise is that testing for initial substring cases only saves approximately 10% of the work no matter how far we go in the enumeration, but the other tests, particularly the conflicting rules test, become more and more productive. Figure 6 is a log-log plot of the performance ratio r versus enumeration index n, providing a visualization of the initial results shown in Table 10.

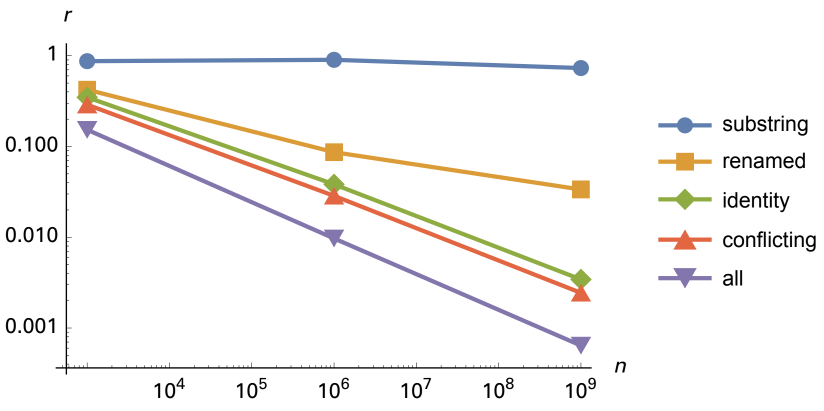

Figure 6. Performance comparison of long-jump tests, showing logarithmically decreasing "work to be done" ratio for higher enumeration indices.

Continued looping through the RSS enumeration, implementing all four tests, extends the trend line shown for the "all" case, which appears as a relatively straight line on the log-log plot. The best-fit trend line is $\ln r = 0.38827 - 0.35551 \ln n$, with a correlation coefficient of 0.9976. Assuming a relatively constant time per case considered, r can be taken as proportional to t / n. Substituting $r = kt / n$ into $\ln r == a - m \ln n$ yields

$$\ln(kt/n) = a - m \ln n \Rightarrow \ln k + \ln t - \ln n = a - m \ln n \Rightarrow \ln t = (a - \ln k) + (1 - m) \ln n \Rightarrow t = \left(e^a / k\right) n^{(1-m)} = K n^{(1-m)},$$

indicating that the time t needed to reach index n in the enumeration is approximately proportional to $n^{(1-m)} \approx n^{(1-0.35551)} \approx n^{0.6}$, a significant improvement over the default $O(n)$ behavior.

7. Conclusion

The new generalized enumeration for substitution system (GSS), derived in Section 3, has been shown to allow spectacular acceleration due to long-jump tests, a benefit that far more than offsets the inclusion of rules of type ""→"", which had been explicitly excluded in the previous enumeration. We used the automatic grouping of similar rulesets in the same weight class together in runs to permit long jumps over the unwanted "nothing-to-nothing" cases and then extended that effort to four tests particularly useful for the efficient treatment of sequential substitution systems. The actual implementation was done for the reduced enumeration substitution system, derived in Section 4, but the techniques described admit of simple generalization to the GSS enumeration as well. Three of the tests considered allow progressively larger jumps for higher indices, with performance improvement of roughly 10 times every 3 orders of magnitude. This translates into an average "acceleration" in the enumeration loop such that calculation time to treat all cases up to index n is of $O\left(n^{0.6}\right)$. While not the hoped-for $O\left(\ln n\right)$, this still represents a huge acceleration over the previous $O(n)$, a most satisfactory result permitted by the regularity and mathematical predictability of the new enumeration.

Acknowledgments

The authors acknowledge the generous support of the Academic Research Committee of Southern Adventist University for funding this project, thank Breanna Bowden-Green, Kelsey Dobbs, Sean Bryant and Amy Beard for previous work on the SSS project, and thank Charles Sarr, Jonathan Ziesmer and Lewis Caviness for helpful suggestions in the present work.

References

[1] S. Wolfram, *A New Kind of Science*, Champaign, IL: Wolfram Media, Inc., 2002. pp. 1–22, 82–94, 433–503.

[2] K. E. Caviness, “Causal Networks of Sequential Substitution Systems: Views of Highly Non-local Systems,” poster session and lecture at Area della Ricerca CNR-Pisa, 2009.

[3] K. E. Caviness, “Indexing Strings and Rulesets: An Exploration Leading to an Enumeration,” *The Mathematica Journal*, **13**, 2011. www.mathematica-journal.com/2011/05/11/indexing-strings-and-rulesets.

[4] K. Caviness. “Universal String Enumeration” from the Wolfram Demonstrations Project–A Wolfram Web Resource. demonstrations.wolfram.com/UniversalStringEnumeration.

[5] K. Caviness. “Tree of Strings” from the Wolfram Demonstrations Project—A Wolfram Web Resource. demonstrations.wolfram.com/TreeOfStrings.