Original Sessie Enumeration:

The first attempt to enumerate sessies was published here:

http://www.mathematica-journal.com/2011/05/indexing-strings-and-rulesets/

This has now been replaced by enumerations allowing better predictions based on simpler mathematical relationships.

Generalized Quinary Enumeration:

Since then we have simplified and generalized the enumeration algorithm (the subject of a paper by Caviness, Case, Morrow & Kratzke*)(2020)*. A new fully general algorithm based on base-5 operation codes was used in the code everywhere for several years. For example,

fromGeneralizedRank[n]

gives the n-th ruleset in the generalized quinary enumeration. The first 62 rulesets are listed below. If we define the weight of a ruleset as the sum of the individual letters, with A=1, B=2, C=3, …, it is clear that in this list, weight 1 rulesets (containing only 1 letter, an A) all appear before weight 2 rulesets (containing either 2 As or 1 B), and weight 2 rulesets before those of weight 3, etc.

fromGeneralizedRankShowSteps /@ Range[62] // Column (* display steps in creating rulesets *)

```
1 : 0₂ ₅ :      {"" -> "A"}
2 : 1₂ ₅ :      {"A" -> ""}
3 : 0₂0₅ :      {"" -> "A", "" -> "", "A" -> ""}
4 : 0₂1₅ :      {"" -> "A", "" -> "A"}
5 : 0₂2₅ :      {"" -> "A", "A" -> ""}
6 : 0₂3₅ :      {"" -> "AA"}
7 : 0₂4₅ :      {"" -> "B"}
8 : 1₂0₅ :      {"A" -> "", "" -> "A"}
9 : 1₂1₅ :      {"A" -> "", "A" -> ""}
10 : 1₂2₅ :     {"A" -> "A"}
11 : 1₂3₅ :     {"AA" -> ""}
12 : 1₂4₅ :     {"B" -> ""}
13 : 0₂00₅ :    {"" -> "A", "" -> "", "A" -> "", "" -> "A"}
14 : 0₂01₅ :    {"" -> "A", "" -> "", "A" -> "", "A" -> ""}
15 : 0₂02₅ :    {"" -> "A", "" -> "", "A" -> "A"}
16 : 0₂03₅ :    {"" -> "A", "" -> "", "AA" -> ""}
17 : 0₂04₅ :    {"" -> "A", "" -> "", "B" -> ""}
18 : 0₂10₅ :    {"" -> "A", "" -> "A", "" -> "", "A" -> ""}
19 : 0₂11₅ :    {"" -> "A", "" -> "A", "" -> "A"}
20 : 0₂12₅ :    {"" -> "A", "" -> "A", "A" -> ""}
21 : 0₂13₅ :    {"" -> "A", "" -> "AA"}
22 : 0₂14₅ :    {"" -> "A", "" -> "B"}
23 : 0₂20₅ :    {"" -> "A", "A" -> "", "" -> "A"}
24 : 0₂21₅ :    {"" -> "A", "A" -> "", "A" -> ""}
25 : 0₂22₅ :    {"" -> "A", "A" -> "A"}
26 : 0₂23₅ :    {"" -> "A", "AA" -> ""}
27 : 0₂24₅ :    {"" -> "A", "B" -> ""}
28 : 0₂30₅ :    {"" -> "AA", "" -> "", "A" -> ""}
29 : 0₂31₅ :    {"" -> "AA", "" -> "A"}
30 : 0₂32₅ :    {"" -> "AA", "A" -> ""}
31 : 0₂33₅ :    {"" -> "AAA"}
32 : 0₂34₅ :    {"" -> "AB"}
33 : 0₂40₅ :    {"" -> "B", "" -> "", "A" -> ""}
34 : 0₂41₅ :    {"" -> "B", "" -> "A"}
35 : 0₂42₅ :    {"" -> "B", "A" -> ""}
36 : 0₂43₅ :    {"" -> "BA"}
37 : 0₂44₅ :    {"" -> "C"}
38 : 1₂00₅ :    {"A" -> "", "" -> "A", "" -> "", "A" -> ""}
39 : 1₂01₅ :    {"A" -> "", "" -> "A", "" -> "A"}
40 : 1₂02₅ :    {"A" -> "", "" -> "A", "A" -> ""}
41 : 1₂03₅ :    {"A" -> "", "" -> "AA"}
42 : 1₂04₅ :    {"A" -> "", "" -> "B"}
43 : 1₂10₅ :    {"A" -> "", "A" -> "", "" -> "A"}
44 : 1₂11₅ :    {"A" -> "", "A" -> "", "A" -> ""}
45 : 1₂12₅ :    {"A" -> "", "A" -> "A"}
46 : 1₂13₅ :    {"A" -> "", "AA" -> ""}
47 : 1₂14₅ :    {"A" -> "", "B" -> ""}
48 : 1₂20₅ :    {"A" -> "A", "" -> "", "A" -> ""}
49 : 1₂21₅ :    {"A" -> "A", "" -> "A"}
50 : 1₂22₅ :    {"A" -> "A", "A" -> ""}
51 : 1₂23₅ :    {"A" -> "AA"}
52 : 1₂24₅ :    {"A" -> "B"}
53 : 1₂30₅ :    {"AA" -> "", "" -> "A"}
54 : 1₂31₅ :    {"AA" -> "", "A" -> ""}
55 : 1₂32₅ :    {"AA" -> "A"}
56 : 1₂33₅ :    {"AAA" -> ""}
57 : 1₂34₅ :    {"AB" -> ""}
58 : 1₂40₅ :    {"B" -> "", "" -> "A"}
59 : 1₂41₅ :    {"B" -> "", "A" -> ""}
60 : 1₂42₅ :    {"B" -> "A"}
61 : 1₂43₅ :    {"BA" -> ""}
62 : 1₂44₅ :    {"C" -> ""}
```

Within any given weight, advancing through the enumeration tends to compact the weight into fewer characters, moving it gradually towards the left. So the most spread out ruleset of weight 3, {"A"->"", ""->"A",""->"","A"->""} is the first of this weight in the list, {"A"->"",""->"B"} appears further down, and {"C"->""} is the last of this weight, with the entire weight compacted to the left.

Note: The "most spread out" rulesets have exactly 2 empty strings between single "A" strings, since we want to include the possibility of deletion rules (where the replacement string is "") and insertion rules (where the search string is ""), and the possibility that an insertion might follow right after a deletion, thus putting 2 empty strings in sequence. E.g., {"A"->"",""->"B"}. But we will never need more than 2 consecutive empty strings, and this algorithm also produces ""->"" rules. More on that problem below.

The algorithm is guaranteed to produce all valid rulesets somewhere in the list, if one goes far enough. It can easily be seen that there is no limit on the number of rules in the ruleset, all cases eventually occurring, and all combinations of all letters will eventually appear everywhere in the ruleset.

As already mentioned, this does generate some unneeded cases, e.g., with empty strings at too many positions in the ruleset, such as ""->"" rules, or multiple ""->*something* rules -- only the first one could ever be used, or indeed any case where an insertion rule appears before the final position in the ruleset. But the most important thing is that all cases we want to study are included, and there is no repetition at all in the enumeration. It is also worth noting that this enumeration never produces invalid rulesets.

Ruleset tests:

Any ruleset that contains a problem can be skipped without generating the sessie and its network, to save time. In addition, the pattern and order of the enumeration lets us skip over multiple cases at once. If such a "long-jump" is possible, the test returns the first ruleset in the enumeration that does not have that particular problem, the target of the long-jump. The main tests are:

Tests that allow long-jumps:

TestForConflictingRules	(* Two rules conflict if the second rule never has a chance to be used because the first rule preempts it. Ex/ {"C"->"", "A"->"B", "AA"->"C"} : rule 3 conflicts with rule 2, so the ruleset is equivalent to {"C"->"", "A"->"B"}, omitting rule 3 *)
TestForNonSoloIdentityRule	(* an identity rule is a rule in which a string is replaced by itself, no further rules will be used if the identity rule ever matches *)
(or TestForIdentityRule)	(* if solo identity rules are treated separately *)
TestForInitialSubstringRule	(* Ex/ {"A"->"BAB", …} If 1st rule matches, it will never fail again. *)
TestForRenamedRuleSet	(* 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. Ex/ {"BAB"→"ABA","A"→"B","B"→"AA"} is indistinguishable from {"ABA"→"BAB","B"→"A","A"→"BB"}
TestForUnusedRules	(* Still under construction: the idea is that if a rule is never used, it could be omitted without affecting the results. The difficulty is in being sure that the rule will never be used, rather than only rarely. *)

Tests that eliminate the ruleset, but don't allow long-jumps:

ShorteningRuleSetQ	(* at least one rule shortens the state string, and none lengthen it, so this will eventually die out *)
UnbalancedRuleSetQ	(* some letter appears on only one side of the rules, so that letter is either created or destroyed, but not both: that letter is not needed for the main pattern of the sessie *)

(*Could we add a short description of what these tests are?) [Done! --KC]*

Camille continued Christen's effort to identify the long jump target ruleset directly based on the quinary (base-5) index of the problem ruleset. The plan was to speed up the tests, as well as to increase understanding of the enumeration. This effort will now be transferred to the new base-3 algorithm (see below).

Reduced Quinary Substitution System enumeration:

Since all rulesets containing an insertion rule (""->*something*) will be discarded by TestForConflictingRules, if the insertion rule is not the last rule, the only way an insertion rule can be the first one is if the ruleset has only the one rule. A solo insertion rule can never delete cells, and so cannot produce a network. Therefore it is logical to eliminate all initial insertion rule cases, and this can easily be done by an adjustment to the enumeration itself, basically dropping the first (binary) digit of the Generalized Quinary enumeration. Here are the first 31 rulesets generated by the reduced SS enumeration are shown here:

```
fromReducedRank /@ Range[31] // Column
```

```
{A →}
{A → , →A}
{A → , A →}
{A → A}
{AA →}
{B →}
{A → , →A, → , A →}
{A → , →A, →A}
{A → , →A, A →}
{A → , →AA}
{A → , →B}
{A → , A → , →A}
{A → , A → , A →}
{A → , A → A}
{A → , AA →}
{A → , B →}
{A → A, → , A →}
{A → A, →A}
{A → A, A →}
{A → AA}
{A → B}
{AA → , →A}
{AA → , A →}
{AA → A}
{AAA →}
{AB →}
{B → , →A}
{B → , A →}
{B → A}
{BA →}
{C →}
```

The RSS enumeration contains the same rulesets as the GSS enumeration in the same order, with initial insertion rules removed. The encoding is also the same, except for the removal of the initial bit that distinguished between rulesets beginning with an empty string and those not.

The decision was made to use the RSS enumeration by default, since initial insertion rules have simple, well-understood behavior.

New Trinary Enumeration:

See notes in 2019 meetings. This will probably supersede the quinary enumerations, since the math is significantly simpler, and the only disadvantage is the appearance of some invalid codes, which are easily jumped over, much as with any long jumps.