Distribution of Degeneracy in the Phenotypic Search Space of Grammatical Evolution Algorithms

Prepared by:

David D Yerger

Faculty Advisors:

Dr. Thomas Montoya

REU Site Director, Department of Electrical and Computer Engineering

Dr. Charles Tolle

Associate Professor, Department of Electrical and Computer Engineering

Dr. Alfred Boysen

Professor, Department of Humanities

Program Information:

National Science Foundation

Grant NSF #1359476

Office of Naval Research (ONR)

Grant ONR N00014-12-1-0347

Research Experience for Undergraduates

Summer 2014

South Dakota School of Mines and Technology

501 E. Saint Joseph Street

Rapid City, SD 57701

Contents

Abstract **3**

1 Introduction **3**

1.1 Background 3

1.2 Objectives 7

2 Broader Impact **7**

3 Procedures **8**

4 Results **10**

5 Discussion **11**

6 Conclusion **14**

6.1 Summary 15

6.2 Recommendations for Future Work 15

References **15**

Acknowledgments **16**

Appendix 1 **17**

Appendix 2 **32**

Appendix 3 **37**

Appendix 4 **40**

Abstract

Grammatical evolution has been used to solve optimization problems and to mathematically model dynamic systems. A key element of grammatical evolution is its grammar. However, a method to quantitatively analyze the degeneracy of grammars has not been proposed. The purpose of this study is to explore the degeneracy of non-trivial grammars and to show the distribution of phenotypes in the phenotypic search space. MATLAB was used to program grammars, generate genes, collect similar phenotypes, and display the distribution of phenotypes. This research has produced a new method for analyzing degeneracy and shown the uneven distribution of phenotypes. Knowing the distribution of degeneracy within a grammar could be used to improve future grammars.

1 Introduction

1.1 Background

Evolutionary Automatic Programming (EAP) refers to "systems that adopt evolutionary computation to automatically generate computer programs" [5]. This means that the goal of EAP is to create a program that solves a problem just from the description of the problem. One example of EAP is Genetic Programming (GP), which has many different variants. Traditionally, GP uses tree-like structures to represent the solution to a problem. A subdivision of GP is Grammar Evolution (GE), which differs from GP in that it has variable length linear genomes, has a mapping from genotype to phenotype, and uses a grammar [5].

The gene in GE is a list of integers, usually written as 8 bit binary numbers. The GE algorithm starts with many randomly generated genes. These genes are then processed through a genotype-phenotype mapping using a grammar. The grammar is usually given in Backus-Naur Form (BNF), which defines the tuple, N, T, P, S, where N is the set of non-terminals, T is the set of terminals, P is the set of production rules, and S is the start symbol [7]. Figure 1 shows an example of a grammar in Backus-Naur Form. In the grammar in Figure 1, $N = \{\text{expr}, \text{op}, \text{func}, \text{digit}\}$, and $T = \{x, y, z, +, -, *, /, \sin, \cos, \exp, \log, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. The production rules are shown in the BNF, and $S = <\text{expr}>$.

The process of mapping the gene onto a phenotype involves the use of modular arithmetic. The gene is read from left to right, and the non-terminals are replaced by terminals until either there are no more non-terminals or the limit for wrapping has been reached. The same gene run through the genotype-phenotype mapping will produce the same expression each time. Wrapping involves going back to the first integer of the gene once the last has been evaluated. Normally, if the wrapping limit is reached and

```
S : : = <expr>  (0)

<expr>  : : = <expr> <op> <expr>      (0)
            | ( <expr> )              (1)
            |  <func> ( <expr> )      (2)
            | <digit>                 (3)
            | x                       (4)
            | y                       (5)
            | z                       (6)

<op>  : : = +   (0)
          | -   (1)
          | *   (2)
          | /   (3)

<func> : : =  sin   (0)
            | cos   (1)
            | exp   (2)
            | log   (3)

<digit> : : = 0  (0)
            | 1 (1)
            | 2 (2)
            | 3 (3)
            | 4 (4)
            | 5 (5)
            | 6 (6)
            | 7 (7)
            | 8 (8)
            | 9 (9)
```

Figure 1: Example of grammar in BNF [9]

the expression is not complete, the individual is designated invalid and is given the worst possible fitness value [7]. The wrapping limit is normally set to 10, but other limits have been used [8]. In some cases, not all integers will be used in order to produce an expression. The selection of rules follows the basic pattern of rule $= V \mathrm{mod} NR$, where V is the integer value in the gene, and NR is the number of rules for the non-terminal [9]. Table 1 shows an example of the genotype to phenotype mapping.

After the initial genes have been mapped into phenotypes, each of the phenotypes is run through a fitness function which gives each gene a fitness value. For example, if the phenotypes are mathematical expressions and this expression is needed to fit a set of points, the sum of the square error could be used to give each gene corresponding to the phenotype a fitness value. Similar to Spencer's survival of the fittest and Darwin's natural selection, the genes which correspond to healthy phenotypes will survive and the less fit genes will be eliminated [5]. The surviving genes will mate to produce offspring by a crossover mechanism, which is demonstrated in Figure 2. Also, there is a probability of mutation, which could

Table 1: Example of genotype to phenotype mapping using grammar in Figure 1

String	Gene	Operation
<expr>	**2**, 4, 4, 8, 9, 2	2 mod 7 = 2
<func> (<expr>)	2, **4**, 4, 8, 9, 2	4 mod 4 = 0
sin(<expr>)	2, 4, **4**, 8, 9, 2	4 mod 7 = 4
$\sin(x)$		

change one integer in the gene. After the population has finished mating, the new population, including the offspring and parents, is put through the same proceedings, which includes genotype to phenotype mapping, fitness evaluation, survival/elimination, and mating, until a sufficient fitness value is achieved or the limit on the number of generations is reached [7].

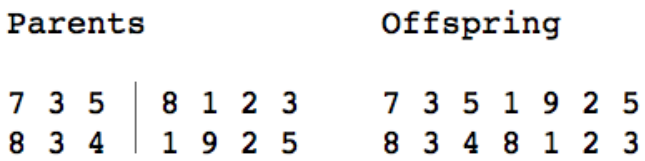


Figure 2: Demonstration of crossover. The vertical line represents the crossover point.

Grammatical Evolution uses the principles in natural biology to mimic life. An advantage of GE is that the search and solution space are separate. This relates more closely to actual biology and allows for the study of the genotype to phenotype mapping. One could see the string of bits as a strand of DNA. Through transcription, DNA is translated into RNA. Transcription can be likened to the bit to integer process. In order to translate RNA into amino acids, the nucleobases are organized into groups of three. These groups of three are called codons and are equivalent to the integers in grammatical evolution. The codons specify the sequence of amino acids in the produced polypeptide chain. The process from codons to amino acids is called translation and can be likened to using the integers to call rules in the grammar. As mentioned previously, not all integers are always used, which is related to introns in biology. The simplification of the expression can be related to production of the protein. This last step may not be directly related to every protein, since certain proteins require multiple polypeptide chains in order to be functional [8]. Figure 3 demonstrates the connection between GE and biology.

Symbolic regression uses grammars that include mathematical functions. Symbolic regression has

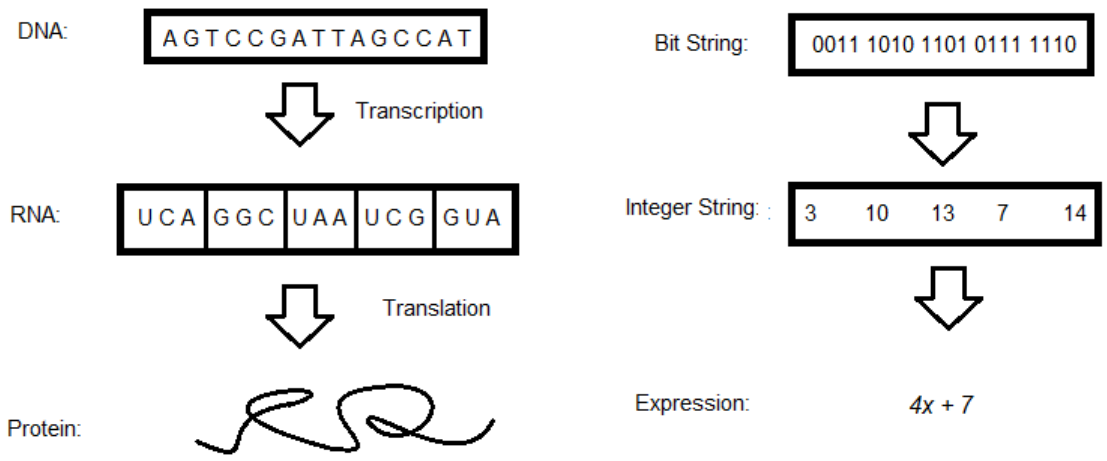


Figure 3: Visualization of the connection between GE and biology

been used to symbolically produce mathematical expressions that fit a set of data on a given domain [5]. Grammars used in symbolic regression can also be used for symbolic integration, solving differential equations [9], and other modeling problems.

Degeneracy occurs when two or more genotypes are mapped to the same phenotype. There is evidence that more degeneracy in EAP increases performance. This improvement is related to the Neutral theory of evolution, which says that degenerate genetic code increases the genetic diversity within a population [4]. Ryan provides an interesting analogy, where he relates duplication of genes to hemoglobin [7]. Hemoglobin is a metalloprotein which consists of protein subunits. Hemoglobin is essential to the transportation of oxygen throughout the body. Ryan sees the production of these subunits as relating to duplications of phenotypes. This analogy can be expanded with degeneracy. Degenerate genetic code allows organisms to survive if a mutation causes a harmful effect, and in some cases, small mutations are actually beneficial to the organism. In parts of Africa, since malaria is common, having a mix of sickle cell and normal hemoglobin cells turns out to be helpful since it lessens the chance of contracting malaria. This could be taken as an example of the benefit of degenerate code. The slight variation in each gene causes the mixing of the different cells to benefit the human.

One researcher, Rothlauf, makes a distinction between redundancies, which is his word for degeneracy. Synonymously redundant means that when a genotype is closely related to another genotype, they will more likely map to similar phenotypes. Nonsynonymously redundant means that when a genotype is closely related to another genotype, they will less likely map to similar phenotypes. This means that with nonsynonymous redundancy, an offspring is more likely to be completely different from the parents. Synonymous redundancy led to improved performance [6].

Although this finding is interesting, the scope of the research is limited to the trivial voting mapping problem. The trivial voting mapping uses binary numbers for the genotypic search space and the number of ones in the binary number for the phenotypic search space. Since binary numbers follow a well known pattern, there is no mystery to the redundancy of this system. The main purpose of using this is to test the amount of redundancy, which can be easily controlled by controlling how many bits are allowed. Other redundancy research in grammatical evolution has made an assumption, that when increasing the bit size the degeneracy increases evenly. Although this is true in trivial cases, this assumption is false for others and can only be used as an approximation for the amount of degeneracy. However, a more precise measure of degeneracy might be appropriate.

1.2 Objectives

The purpose of this research is to develop a method to explore the degeneracy of grammars. With this developed technique, the distribution of degeneracy is revealed and could be compared with different grammars. This could lead to revealing the reasons for the different distributions.

2 Broader Impact

Revealing the distribution of degeneracy of grammars is important in understanding how GE works. GE is used in a multitude of applications. GE can be used to produce programs that solve various problems, and could ultimately lead to improved artificial intelligence. Although this study deals only with symbolic regression, the findings likely hold for other non-trivial applications.

This study has revealed that certain assumptions about the distribution cannot be made, for example, assuming that every phenotype is assigned to at least one genotype, and that the distribution is even. This study has shown that, with certain grammars, there is a limit to the size of the phenotypic search space, which can be a rather disturbing finding. The implication is that the GE algorithm may be searching a phenotypic search space that does not contain the best answer. Knowing this, grammars could be designed to increase the size of the phenotypic search space.

This study has revealed that the factors that affect the size of the search space are the gene length, wrapping, the number of non-terminals, the number of terminals per non-terminal, reading from the right or left, and the terminating process. It is known that there are deep connections between degeneracy and performance of GE algorithms. Understanding the distribution of degeneracy could allow for better grammars to be written and consequentially better performance in GE algorithms.

3 Procedures

The first step was to program grammars using MATLAB. The grammars in Figures 1 and 4 were programmed. Code for these grammars can be seen in the Appendix 1. Tsoulos grammar was used for solving differential equations. The second grammar is a basic polynomial grammar.

```
S : : = <expr>  (0)

<expr>  : : = <expr> <op> <expr>       (0)
              | ( <expr> <op> <expr> ) (1)
              |  <var> ^ <const>       (2)
              | <const> * <expr>       (3)

<op>  : : = +   (0)
          | -   (1)
          | *   (2)

<var> : : =   x       (0)
              | y     (1)

<const> : : = 1  (0)
            | 2  (1)
            | 3  (2)
            | 4  (3)
```

Figure 4: BNF Basic Polynomial grammar [2].

To program these grammars, the process was split into two functions. The first function is "build_tree," which creates a parse tree according to the specific grammar. These parse trees contain forward and backward pointers, which allow the second function, "build_string," to create the mathematical expressions. Also, an evaluation function was written in order to eliminate the complex and infinite expression. These three functions were then used in a grammar code, which creates genes of a specified length with different integers, runs the "build_tree," "build_string," and evaluation function on one gene in parallel, and creates a list of the mathematical expressions. The mathematical expressions are then compared to each other and gathered into bins. This produces a distribution of the degeneracy produced from running all the genes. This distribution is then plotted in a histogram.

The string comparison function in MATLAB was computationally expensive, so in order to make this process quicker, a couple solutions were attempted. First, the string comparison was written in parallel

by first splitting the phenotypes into groups of two and comparing, then grouping two of the compressed groups of two and comparing the first group to the second. This was repeated until only one group remained. Although this did shorten the run time, it was not sufficient for longer gene lengths. Next, for the polynomial grammar, points were compared according to the degree of the polynomial. This was computationally inexpensive compared to the original string comparison, but could not be used for the other grammar. The final solution converted the mathematical strings into a list of integers using the double function in MATLAB and compared these lists of integers. This was also computationally inexpensive, and was sufficient for both grammars. It should be noted that although this comparison technique was sufficient, there likely exists a method that is computationally superior. Also, the eval function in MATLAB only simplifies functions by combining integers. This could cause two of the same functions to be considered different. For example, $x^2 + x$ would be different from $x \cdot (x + 1)$, even though they are the same expression. The simplify function in MATLAB could likely solve this problem. Even though this is a flaw, this actually means that the grammars are even more degenerate than what has been shown. The difference was considered negligible in this study. The code for the comparison techniques can be found in the Appendix 2.

One major difference between this research and other studies in GE is that the gene length was set for a specific run. Also, the gene lengths considered are not as long as normal. The largest gene length considered was 10, which is relatively small. When the amount of genes exceeded 100,000, a sample of 100,000 random genes were chosen and plotted. This was done to shorten the run-time, and was considered sufficient to support this paper's claim. Also, when running trials from the right, with gene length greater than 7, and wrapping limit of 10, a depth error occurred, which is caused when an expression has too many nested parentheses. This problem has not been solved, but may be one of the major reasons to transfer this process to some other language. Perhaps one could solve this problem in MATLAB, but it might be more beneficial to try something different. One final consideration is the choice to run only combinations of 1 through the highest number of terminals per non-terminal. Doing this creates inherent degeneracy because of the modular arithmatic when considering the non-terminals with less than the highest number of terminals per non-terminal. Although this will cause certain functions to be more degenerate than others, the degree of degeneracy that would be caused by this inherent degeneracy would be less than what has been shown. Also, since the integers are normally 8 bits, there would be even more degeneracy. Specifically, for a gene length equal to n, the degeneracy would be 64^n times more than what has been shown for the polynomial grammar. Also, two different cases when the wrapping limit is reached (terminating processes) were considered. First, the expression was made invalid if the tree still contained non-terminals when the wrapping limit was reached. Second, instead of making the expression

invalid, the rules were switched to complete the expression with terminals. This second method is similar to the method used in the Multi-Chromosomal GE study [1].

In order to visualize the results, a histogram function was written. The histogram shows the percent of generated symbols versus the number of symbol replications. Ideally, if the distribution of degeneracy was evenly distributed, there would be only one bar at 100%. Also, some statistics were run including the mean, standard deviation, and entropy. The formula for entropy is $-\sum_{i=1}^{n} p(x_i) \log_2 p(x_i)$, where $p(x_i)$ is the probability of an event x_i. The programs for these measures can be found at the end of Appendix 2.

The last step is to compare the differences in the distributions and to propose reasons for these differences from the grammars. The factors that were explored were the gene length, wrapping, the number of non-terminals, the number of terminals per non-terminal, reading from the left or right, and the terminating process.

4 Results

As seen in Figures 5 and 6, the distribution of degeneracy is not evenly distributed. The distributions are also very different. Along the top of the histogram, the number of phenotypes for each bar is also given. The histograms for different trials can be seen in the Appendix 4. The naming system for these histograms use the form Left/Right, gene length, wrapping limit, valid/invalid. For example, L41V was read from left, had a gene length of 4, had a wrapping limit of 1, and used the terminating rule which completed the tree making valid expressions. All histograms shown used the polynomial grammar. The Tsoulos grammar was as degenerate as the polynomial grammar. Also, since the Tsoulos grammar has terminals with the starting symbol, this introduces a very large amount of degeneracy for the expressions x, y, and z. These expressions are coded by just 1 codon, which leaves the rest of the codons unexpressed in the algorithm. There was no trial that had a perfectly even distribution, and the distributions did change for each trial. Table 2 shows the statistics from the trials with the terminating rule, and Table 3 shows some statistics from the tree completion trials. More statistics are available in Appendix 3 for the tree completion trials. Upon comparison, it is apparant that the tree completion has a lower mean and a higher percentage of phenotypes. This implies that the tree completion method is less degenerate than the terminating rule. This is understandable because the expressions that require less codons to be completed are inherently degenerate. Although the tree completion method is less degenerate, it cannot be concluded that this improves the efficiency of the algorithm. Perhaps the invalid expressions in the terminating rule grammar would have lower fitness values than the valid expressions. This could be explored in a study of the efficiency of these grammars, but was not explored in this study.

Figure 6 shows a comparison of reading from the left or right. For most cases, more unique expressions were evolved from reading from the right. Also, more phenotypes were created from reading from the right. This could potentially increase the efficiency of the GE algorithm, but has not been studied.

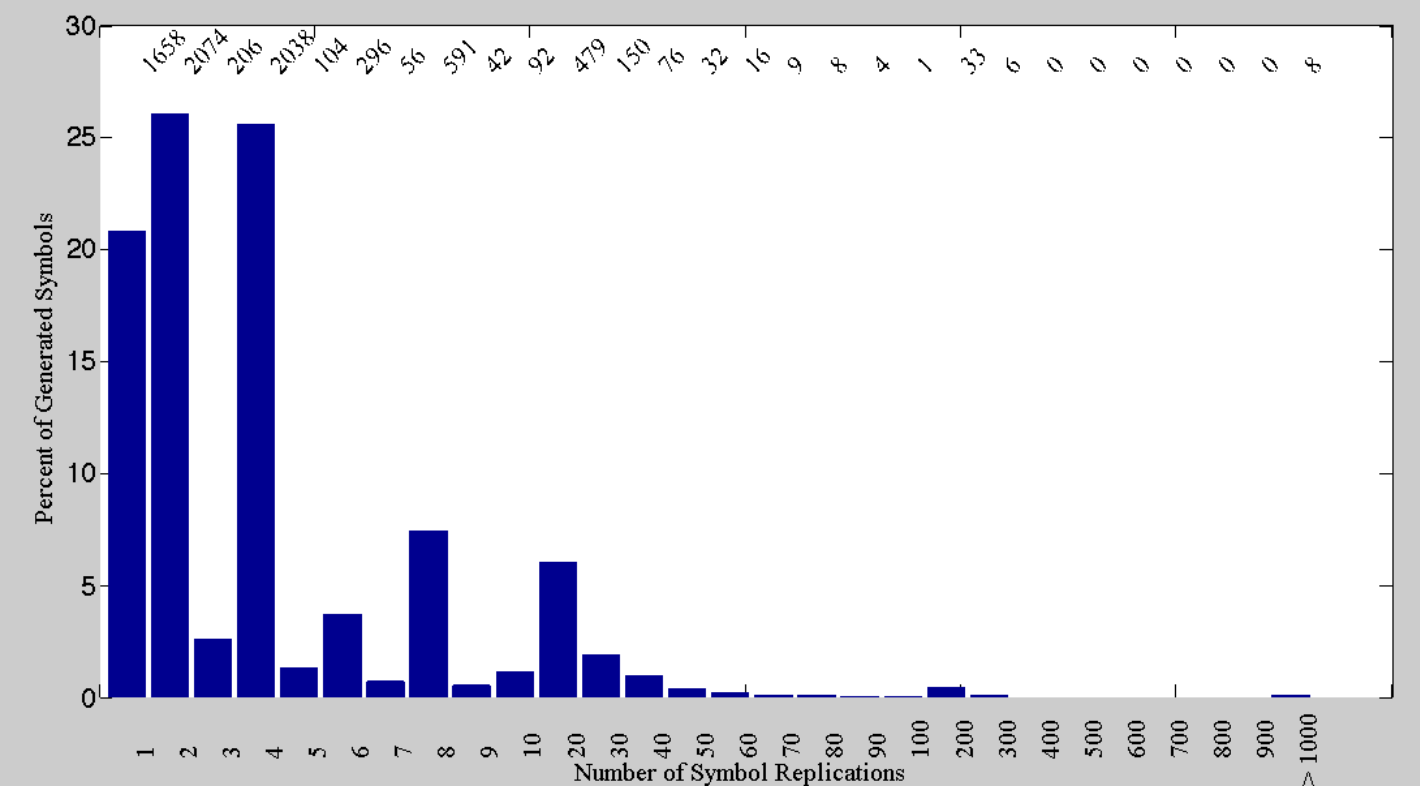


Figure 5: Trial for polynomial grammar, reading from right, gene length of 8 integers, and wrapping limit of 1

It is apparent that the grammar biases the phenotypic search space toward certain expressions. This could have a negative effect on the grammatical evolution algorithm, since, if the biased expressions are not functionally fit, they would clutter the search spaces.

5 Discussion

Although it has been shown that degeneracy is beneficial to GE algorithms, the degree of the uneven distribution has never been revealed. The uneven distribution causes a bias toward the more highly repeated phenotypes. In the case when these phenotypes are relatively fit, this could be beneficial, but, in the case when these phenotypes are relatively unfit, this could be harmful since the unfit phenotypes would clutter the search space. The histogram of the ideal grammar would have only one bar at 100%.

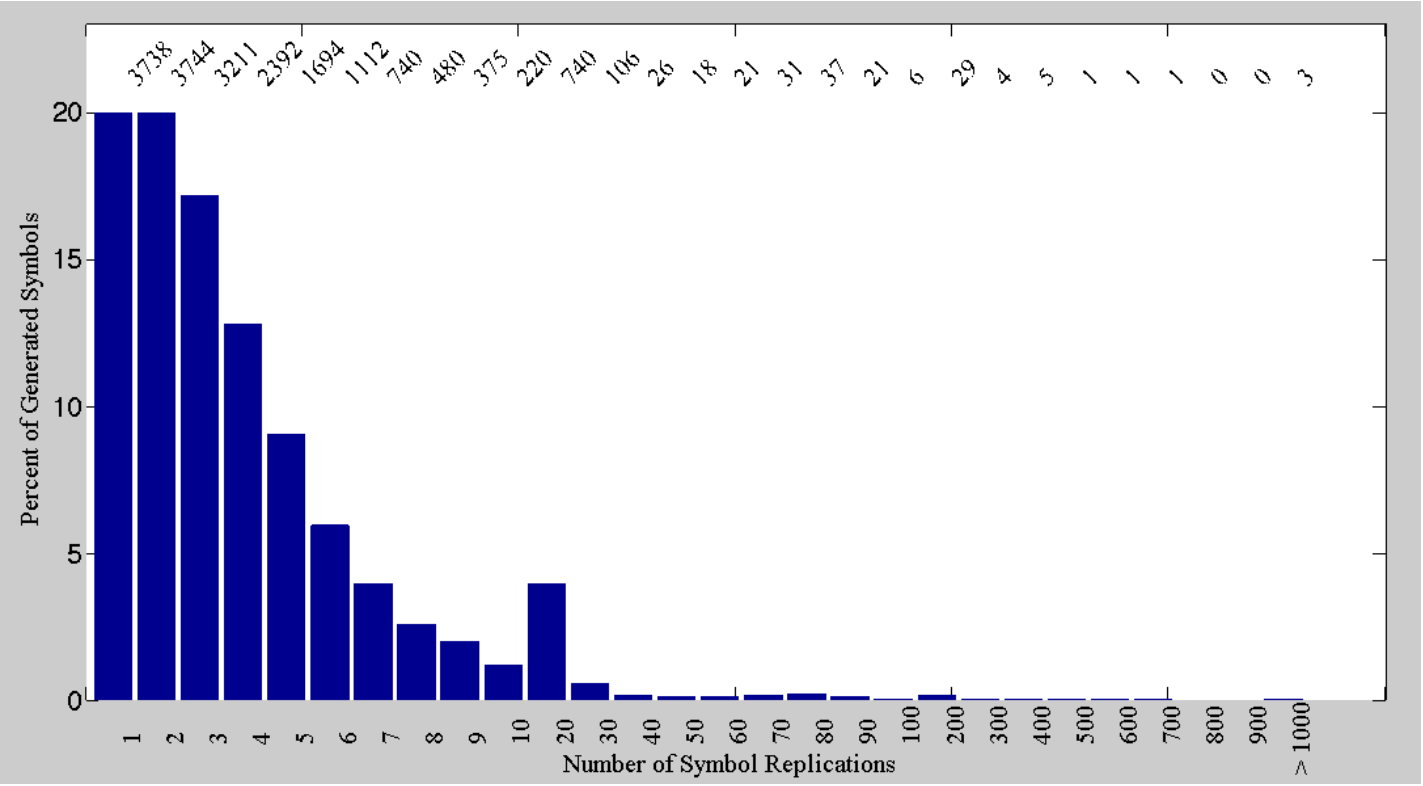


Figure 6: Trial for polynomial grammar, reading from right, gene length of 10 integers, and wrapping limit of 10.

Table 2: Statistics from polynomial grammar with terminating rule

Left or Right	Gene Length	Wrapping Limit	Mean	Standard Deviation	Entropy	Invalid Expressions	Phenotypes	Genotypes	Percent
L	4	10	7.76	6.75	3.04	102	33	256	12.89
R	4	10	6.24	6.41	3.23	97	41	256	16.02
L	5	10	18.29	16.60	2.03	667	56	1024	5.47
R	5	10	18.29	15.97	2.42	605	56	1024	5.47
L	6	10	19.69	29.54	3.27	2078	208	4096	5.08
R	6	10	18.96	29.46	3.07	2179	216	4096	5.27
L	7	10	40.26	80.65	2.50	10107	407	16384	2.48
R	7	10	29.10	68.14	2.92	9463	563	16384	3.44
L	8	10	47.08	172.65	3.57	65536	1392	65536	2.12
R	8	10	33.49	145.90	3.55	34589	1957	65536	2.99

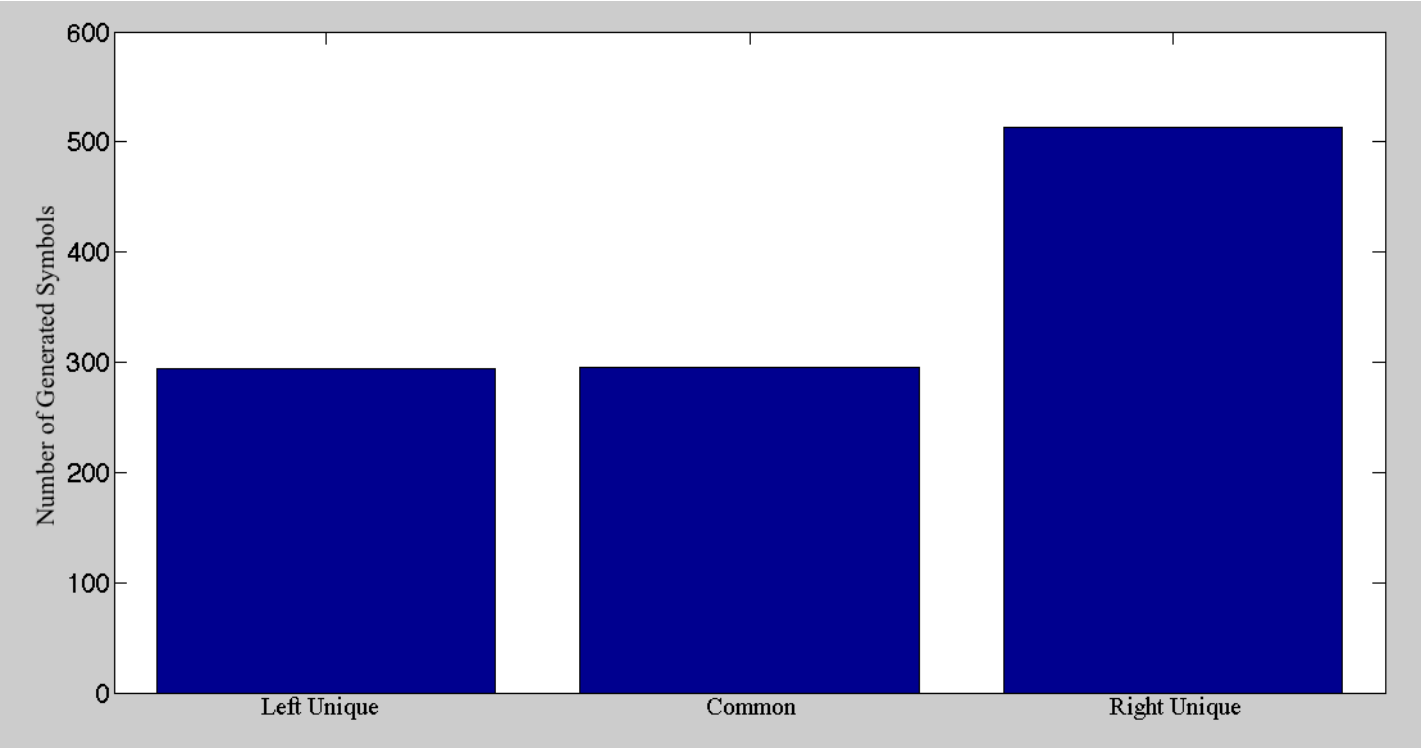


Figure 7: Comparison of phenotypes evolved from reading from the left or right of the polynomial grammar with a gene length of 6 and wrapping of 1.

Table 3: Statistics from polynomial grammar with tree completion

Left or Right	Gene Length	Wrapping Limit	Mean	Standard Deviation	Entropy	Phenotypes	Genotypes	Percent
L	4	10	3.20	4.18	5.71	80	256	31.25
R	4	10	2.15	3.78	6.07	119	256	46.48
L	5	10	2.38	6.92	7.25	431	1024	42.09
R	5	10	2.56	7.04	7.17	400	1024	39.06
L	6	10	4.81	15.09	8.25	851	4096	20.78
R	6	10	2.57	11.26	8.74	1595	4096	38.94
L	7	10	2.36	21.36	10.12	6933	16384	43.32
R	7	10	1.96	18.88	10.48	8376	16384	51.12
L	8	10	6.73	65.94	10.54	9743	65536	14.87
R	8	5	2.60	40.66	11.45	25195	65536	38.44

This would make each solution as likely to be selected at random and would allow easier maneuvering throughout the phenotypic search space.

There likely is no non-trivial grammar that can achieve this ideal case. However, certain grammars may come closer to this ideal than others. This same method could be used to study many grammars and perhaps reveal what conditions bring a grammar closer to this ideal.

One reason for the uneven distribution could be effective genome length. The actual genome length is how many codons are in the gene, and the effective genome length is how many codons are actually used to create the phenotype. It has been shown that the effective genome length is shorter than the actual genome length, which is understandable [3]. The integers that are not used can be related to introns in biology with the difference that introns are not all grouped together at the end. An expression that uses less integers will have a higher degree of degeneracy since the remaining integers do not matter. This problem might be solved by changing the grammar.

For example, using the polynomial grammar, there is a specific set of genes which will produce expressions of the form $k \cdot x_n^l$ where $k = \{1, 2, 3, 4\}$, $l = \{1, 2, 3, 4\}$, and $n = \{1, 2\}$. The gene sequence to produce this type of expression is [4 (1-4) 3 (1-2) (1-4) ...]. This means that there are 32 genes of this form when the gene length is 5. If the gene length is n, and $n \geq 5$, the remaining integers do not matter so there will be $32 \cdot 4^{n-5}$ genes that produce similar expressions. The calculation, $\frac{32 \cdot 4^{n-5}}{4^n} = \frac{32}{4^5} = 3.125\%$, shows that at least 3.125 % of the expressions will be of this form. This analysis doesn't take into account other ways of getting this type of expression, for example, $x_1 + x_1 = 2 \cdot x_1$. Other limited length phenotypes could be analyzed the same way.

6 Conclusion

The assumption that the distribution of the phenotypic search is evenly distributed is false for the typical non-trivial grammars tested, and likely false for most non-trivial grammars. A new method using MATLAB has been developed to study degeneracy on a more quantitative level. Many grammars, especially the grammars involved with symbolic regression, could be studied using the techniques developed. Further study of the effects on grammar variation could lead to improvements in writing grammars, which could lead to improvements in grammatical evolutions results and speed.

6.1 Summary

It has been shown for a basic symbolic regression grammar that the distribution of the phenotypic search space is unevenly distributed. This work was performed using MATLAB, by programming grammars and running different trials. The conclusion reached with the grammars studied is likely able to be extended to all non-trivial grammars. This has implications on the effectiveness of grammars to span a large phenotypic search space. Also, the bias introduced by the grammar causes a chance of cluttering up the search space.

6.2 Recommendations for Future Work

More trials with different grammars, gene lengths, and wrapping limits should be run. Also, other grammars that do not deal with symbolic regression could be studied. In order to firmly conclude that certain conditions are the causes of uneven distribution, more symbolic regression grammars would have to be studied.

References

[1] Akira Hara, Tomohisa Yamaguchi, Takumi Ichimura, and Tetsuyuki Takahama. Multi-chromosomal grammatical evolution. In *Proceedings of 4th International Workshop on COMPUTATIONAL INTELLIGENCE & APPLICATIONS, Okayama, Japan*, 2008.

[2] Jeff S McGough, Alan W Christianson, and Randy C Hoover. Symbolic computation of lyapunov functions using evolutionary algorithms. In *Proceedings of the 12th IASTED International Conference*, volume 697, page 508, 2010.

[3] Michael O'Neill and Conor Ryan. Under the hood of grammatical evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1143–1148, 1999.

[4] Michael O'Neill and Conor Ryan. Grammatical evolution. In *IEEE Transactions on Evolutionary Computation*, volume 5. IEEE, 2001.

[5] Michael O'Neill and Conor Ryan. *Grammatical evolution: evolutionary automatic programming in an arbitrary language*, volume 4. Springer, 2003.

[6] Franz Rothlauf and David E Goldberg. Redundant representations in evolutionary computation. *Evolutionary Computation*, 11(4):381–415, 2003.

[7] Conor Ryan, JJ Collins, and Michael O'Neill. Grammatical evolution: Evolving programs for an arbitrary language. In *Genetic Programming*, pages 83–96. Springer, 1998.

[8] Conor Ryan and Michael O'Neill. How to do anything with grammars. In *Proc. of the Bird of a Feather Workshops, Genetic and Evolutionary Computation Conference*, pages 116–119, 2002.

[9] Ioannis G Tsoulos and Isaac E Lagaris. Solving differential equations with genetic programming. *Genetic Programming and Evolvable Machines*, 7(1):33–54, 2006.

Acknowledgments

This work was made possible by the National Science Foundation REU Site: Bringing Us Together, Improving Communications and Lives grant EEC-1359476 and by Office of Naval Research (ONR) grant N00014-12-1-0347. Thanks to advisor Dr. Charles Tolle and REU site director Dr. Thomas Montoya for their direction and guidance, Professor of English Dr. Alfred Boysen for his critique in writing and speaking, Professor of Mathematics at Moravian College Dr. Michael Fraboni for his encouragement and letter of recommendation, and a special thanks to all of the faculty and students in the Electrical and Computer Engineering department and the Controls Lab for their help.

Appendix 1

```
function rs = build_strpolyparallel(node, tree)

syms x1 x2 y0 y1 y2 y3
%   o is the operation definition
%   s is the variable definition
%   d is the digit definition
o{1}.f = inline('x + y','x','y');
o{2}.f = inline('x - y','x','y');
o{3}.f = inline('x * y','x','y');
o{4}.f = inline('x ^ y','x','y');
% symbol names
s{1} = 'x1';
s{2} = 'x2';

%digits
d{1} = 'y0';
d{2} = 'y1';
d{3} = 'y2';
d{4} = 'y3';
%Because the char function gets upset when a digit is passed through it,
%I've switched the digits for symbols to created the strings. Switched back
%in toeval function
%variable from <expr>
if tree{node}.ln == 0 && tree{node}.ev==1,
    rs = char(s{tree{node}.type});
end

%<var>^<const>
if tree{node}.ln==2 && tree{tree{node}.l(1)}.ev == 3
    t1 = s{tree{tree{node}.l(1)}.type};
    t2 = d{tree{tree{node}.l(2)}.type};
    rs = char(o{4}.f(eval(t1), eval(t2)));
end

%<const>*<expr>
if tree{node}.ln==2 && tree{tree{node}.l(1)}.ev == 4
    t1 = d{tree{tree{node}.l(1)}.type};
    t2 = build_strpolyparallel(tree{node}.l(2), tree);
    rs = char(o{3}.f(eval(t1), eval(t2)));
end

%<expr> <op> <expr>
if tree{node}.ln==3 && tree{node}.parenthesis == 0
    t1 = build_strpolyparallel(tree{node}.l(1), tree);
    t2 = build_strpolyparallel(tree{node}.l(3), tree);
    %the digit 0 would be returned if not for this statement
    %for example eval(x1-x1) ==> 0
    if eval(t1)==eval(t2) && tree{tree{node}.l(2)}.type==2
        rs = char(d{1});
    else
        rs = char(o{tree{tree{node}.l(2)}.type}.f(eval(t1), eval(t2)));
    end
```

```
end
%(<expr><op><expr>)
if tree{node}.ln==3 && tree{node}.parenthesis == 1
    t1 = build_strpolyparallel(tree{node}.l(1), tree);
    t2 = build_strpolyparallel(tree{node}.l(3), tree);
    %the digit 0 would be returned if not for this statement
    %for example eval(x1-x1) ==> 0
    if eval(t1)==eval(t2) && tree{tree{node}.l(2)}.type==2
        rs = char(d{1});
    else
        rs = char((o{tree{tree{node}.l(2)}.type}.f(eval(t1), eval(t2))));
    end
end
return
```

```
Error using build_strpolyparallel (line 24)
Not enough input arguments.
```

Published with MATLAB® R2014a

```
function rs = build_treepolyparallel(gene,mst,nnt,no,nv,nd)


% t is the tree
%   t{n}.ev is the type of event
%   t{n}.type is the index of certain nonterminal's terminals
%   t{n}.ln is the number of daughter leaves.
%   t{n}.mb is the mother leaf for this daughter leaf (backwards pointer).
%   t{n}.branch is the index of the branch off of the nonterminal
%   t{n}.l(nn) is the forward pointer to the daughter leaves.
%   backward pointer is made when branch is created.
%   foreward pointer is made when branch is evaluated.
% n is the index of the leaf we're on
% nn is the index of the subleaf
% mst is the number of wrappings allowed
% nv is the number of variables
% I am using a "reading from left" approach
% modder is the number of terminals for the non-terminal
%ne =    4;  % number of events
%nnt =    4;  % types of nonterminals
%no =    3;  % number of operations
%nd= 4; %number of digits
%nv = 2; %number of variables
counter=0;
n=1;
nn=1;
modder=nnt;
%Using the starting terminal
t{n}.ev=1;
%Gives the length of gene
gl = length(gene);
%This for loop is to allow the second statement in the while loop to
%actually be false.  The only question with doing this for loop is
%How many branches must be covered? 100 is plenty, but what if you
%have a very branched tree. Definitely something to consider for huge sets
for l=2:100
    t{l}.ev=-2;
    t{l}.modder=0;
end
%counter<=mst*gl limits the amount of times we can wrap
%the or statement makes sure that we actually have a connected leaf
while (counter < mst*gl && (t{n}.ev==1 || t{n}.ev==2 || t{n}.ev==3 ||...
        t{n}.ev==4))
    %reads gene from left to right with wrapping (if allowed)
    switch mod(gene(mod(counter, gl)+1), modder)+1
        case 1
            switch (t{n}.ev)
                case 1
                    %In order to read the left most branch the already
                    %existing branches (to the right) must be incremented
                    %by the number of leaves made.
                    for pr=nn:-1:n+1,
```

```
                t{pr+3}.ev=t{pr}.ev;
                t{pr+3}.ln=t{pr}.ln;
                t{pr+3}.modder=t{pr}.modder;
                t{pr+3}.mb=t{pr}.mb;
                t{pr+3}.branch=t{pr}.branch;
            end
            t{n}.ln=3;
            t{n}.parenthesis=0;
            %create daughter leaves
            t{n+1}.ev=1;
            %reference mother leaf
            t{n+1}.mb=n;
            %create unknown number of granddaughter leaves
            t{n+1}.ln=-1;
            %set up modder for each nonterminal
            t{n+1}.modder=nnt;
            %Which branch is this branch?
            t{n+1}.branch=1;
            t{n+2}.ev=2;
            t{n+2}.mb=n;
            t{n+2}.ln=-1;
            t{n+2}.modder=no;
            t{n+2}.branch=2;
            t{n+3}.ev=1;
            t{n+3}.mb=n;
            t{n+3}.ln=-1;
            t{n+3}.modder=nnt;
            t{n+3}.branch=3;
            %increment number of leaves by how many were created
            nn=nn+3;
        case 2
            %operation is addition
            t{n}.type=1;
            t{n}.ln=0;
        case 3
            %variable is x1
            t{n}.type=1;
            t{n}.ln=0;
        case 4
            %digit is 1
            t{n}.type=1;
            t{n}.ln=0;
    end
case 2
    switch (t{n}.ev)
        case 1
            for pr=nn:-1:n+1,
                t{pr+3}.ev=t{pr}.ev;
                t{pr+3}.ln=t{pr}.ln;
                t{pr+3}.modder=t{pr}.modder;
                t{pr+3}.mb=t{pr}.mb;
                t{pr+3}.branch=t{pr}.branch;
            end
            t{n}.ln=3;
```

```
            t{n}.parenthesis=1;
            t{n+1}.ev=1;
            t{n+1}.mb=n;
            t{n+1}.ln=-1;
            t{n+1}.modder=nnt;
            t{n+1}.branch=1;
            t{n+2}.ev=2;
            t{n+2}.mb=n;
            t{n+2}.ln=-1;
            t{n+2}.modder=no;
            t{n+2}.branch=2;
            t{n+3}.ev=1;
            t{n+3}.mb=n;
            t{n+3}.ln=-1;
            t{n+3}.modder=nnt;
            t{n+3}.branch=3;
            nn=nn+3;
        case 2
            %the operation is subtraction
            t{n}.type=2;
            t{n}.ln=0;
        case 3
            %the variable is x2
            t{n}.type=2;
            t{n}.ln=0;
        case 4
            %the digit is 2
            t{n}.type=2;
            t{n}.ln=0;
    end
case 3
    switch (t{n}.ev)
        case 1
            for pr=nn:-1:n+1,
               t{pr+2}.ev=t{pr}.ev;
               t{pr+2}.ln=t{pr}.ln;
               t{pr+2}.modder=t{pr}.modder;
               t{pr+2}.mb=t{pr}.mb;
               t{pr+2}.branch=t{pr}.branch;
            end
            t{n}.ln=2;
            t{n+1}.ev=3;
            t{n+1}.mb=n;
            t{n+1}.ln=-1;
            t{n+1}.modder=nv;
            t{n+1}.branch=1;
            t{n+2}.ev=4;
            t{n+2}.mb=n;
            t{n+2}.ln=-1;
            t{n+2}.modder=nd;
            t{n+2}.branch=2;
            nn=nn+2;
        case 2
            %operation is multiplication
```

```
                    t{n}.type=3;
                    t{n}.ln=0;
                case 4
                    %digit is 3
                    t{n}.type=3;
                    t{n}.ln=0;
            end
        case 4
            switch (t{n}.ev)
                case 1
                    for pr=nn:-1:n+1,
                       t{pr+2}.ev=t{pr}.ev;
                       t{pr+2}.ln=t{pr}.ln;
                       t{pr+2}.modder=t{pr}.modder;
                       t{pr+2}.mb=t{pr}.mb;
                       t{pr+2}.branch=t{pr}.branch;
                    end
                    t{n}.ln=2;
                    t{n+1}.ev=4;
                    t{n+1}.mb=n;
                    t{n+1}.ln=-1;
                    t{n+1}.modder=nd;
                    t{n+1}.branch=1;
                    t{n+2}.ev=1;
                    t{n+2}.mb=n;
                    t{n+2}.ln=-1;
                    t{n+2}.modder=nnt;
                    t{n+2}.branch=2;
                    nn=nn+2;
                case 4
                    %digit is 4
                    t{n}.type=4;
                    t{n}.ln=0;
            end
    end
    %forward pointer (don't need pointer for starting nonterminal)
    if n>1,
        t{t{n}.mb}.l(t{n}.branch)=n;
    end
    %move onto next leaf
    n=n+1;
    %update the current modder, which corresponds to specific leaf
    modder=t{n}.modder;
    %move to next number in gene
    counter = counter + 1;
end
%We must only check to see if the
%rest of the leaves require any terminals.
for k=n:nn,
    if t{k}.ln~=0
        modder=t{k}.modder;
        if t{k}.ev==1
            modder = nv;
        end
```

```
switch mod(gene(mod(counter, gl)+1), modder)+1
case 1
    switch (t{k}.ev)
        case 1
            %the variable is x1
            t{k}.type=1;
            t{k}.ln=0;
        case 2
            %operation is addition
            t{k}.type=1;
            t{k}.ln=0;
        case 3
            %the variable is x1
            t{k}.type=1;
            t{k}.ln=0;
        case 4
            %digit is 1
            t{k}.type=1;
            t{k}.ln=0;
    end
case 2
    switch (t{k}.ev)
        case 1
            %the variable is x2
            t{k}.type=2;
            t{k}.ln=0;
        case 2
            %the operation is subtraction
            t{k}.type=2;
            t{k}.ln=0;
        case 3
            %then variable is x2
            t{k}.type=2;
            t{k}.ln=0;
        case 4
            %the digit is 2
            t{k}.type=2;
            t{k}.ln=0;
    end
case 3
    switch (t{k}.ev)
        case 2
            %operation is multiplication
            t{k}.type=3;
            t{k}.ln=0;
        case 4
            %digit is 3
            t{k}.type=3;
            t{k}.ln=0;
    end
case 4
    switch (t{k}.ev)
        case 4
            %digit is 4
```

```
                    t{k}.type=4;
                    t{k}.ln=0;
                end
            end
            t{t{k}.mb}.l(t{k}.branch)=k;
            counter=counter+1;
        end
    end
end
rs = t;
return
```

```
        Error using build_treepolyparallel (line 27)
        Not enough input arguments.
```

Published with MATLAB® R2014a

```
function rs = build_treepprightinvalid(gene,mst,nnt,no,nv,nd)


% t is the tree
%   t{n}.ev is the type of event
%   t{n}.type is the index of certain nonterminal's terminals
%   t{n}.ln is the number of daughter leaves.
%   t{n}.mb is the mother leaf for this daughter leaf (backwards pointer).
%   t{n}.branch is the index of the branch off of the nonterminal
%   t{n}.l(nn) is the forward pointer to the daughter leaves.
%   backward pointer is made when branch is created.
%   foreward pointer is made when branch is evaluated.
% n is the index of the leaf we're on
% nn is the index of the subleaf
% mst is the number of wrappings allowed
% nv is the number of variables
% I am using a "reading from right" approach
% modder is the number of terminals for the non-terminal
%ne =    4;  % number of events
%nnt =    4;  % types of nonterminals
%no =    3;  % number of operations
%nd= 4; %number of digits
%nv = 2; %number of variables
counter=0;
n=1;
nn=1;
modder=nnt;
%Using the starting terminal
t{n}.ev=1;
%Gives the length of gene
gl = max(size(gene));
%This for loop is to allow the second statement in the while loop to
%actually be false. The only question with doing this for loop is
%How many branches must be covered? 100 is plenty, but what if you have a
%very branched tree. Definitely something to consider for large sets.
for l=2:100
    t{l}.ev=-2;
    t{l}.modder=0;
end
%counter<=mst*gl limits the amount of times we can wrap
%the or statement makes sure that we actually have a connected leaf
while (counter < mst*gl && (t{n}.ev==1 || t{n}.ev==2 || t{n}.ev==3 ||...
        t{n}.ev==4))
    %reads gene from right to left with wrapping (if allowed)
    switch mod(gene(mod(counter, gl)+1), modder)+1
        case 1
            switch (t{n}.ev)
                case 1
                    %In order to read the left most branch the already
                    %existing branches (to the right) must be incremented
                    %by the number of leaves made.
                    for pr=nn:-1:n+1,
```

```
                t{pr+3}.ev=t{pr}.ev;
                t{pr+3}.ln=t{pr}.ln;
                t{pr+3}.modder=t{pr}.modder;
                t{pr+3}.mb=t{pr}.mb;
                t{pr+3}.branch=t{pr}.branch;
            end
            t{n}.ln=3;
            t{n}.parenthesis=0;
            %create daughter leaves
            t{n+1}.ev=1;
            %reference mother leaf
            t{n+1}.mb=n;
            %create unknown number of granddaughter leaves
            t{n+1}.ln=-1;
            %set up modder for each nonterminal
            t{n+1}.modder=nnt;
            %Which branch is this branch?
            t{n+1}.branch=1;
            t{n+2}.ev=2;
            t{n+2}.mb=n;
            t{n+2}.ln=-1;
            t{n+2}.modder=no;
            t{n+2}.branch=2;
            t{n+3}.ev=1;
            t{n+3}.mb=n;
            t{n+3}.ln=-1;
            t{n+3}.modder=nnt;
            t{n+3}.branch=3;
            %increment number of leaves by how many were created
            nn=nn+3;
        case 2
            %operation is addition
            t{n}.type=1;
            t{n}.ln=0;
        case 3
            %variable is x1
            t{n}.type=1;
            t{n}.ln=0;
        case 4
            %digit is 1
            t{n}.type=1;
            t{n}.ln=0;
    end
case 2
    switch (t{n}.ev)
        case 1
            for pr=nn:-1:n+1,
                t{pr+3}.ev=t{pr}.ev;
                t{pr+3}.ln=t{pr}.ln;
                t{pr+3}.modder=t{pr}.modder;
                t{pr+3}.mb=t{pr}.mb;
                t{pr+3}.branch=t{pr}.branch;
            end
            t{n}.ln=3;
```

```
            t{n}.parenthesis=1;
            t{n+1}.ev=1;
            t{n+1}.mb=n;
            t{n+1}.ln=-1;
            t{n+1}.modder=nnt;
            t{n+1}.branch=1;
            t{n+2}.ev=2;
            t{n+2}.mb=n;
            t{n+2}.ln=-1;
            t{n+2}.modder=no;
            t{n+2}.branch=2;
            t{n+3}.ev=1;
            t{n+3}.mb=n;
            t{n+3}.ln=-1;
            t{n+3}.modder=nnt;
            t{n+3}.branch=3;
            nn=nn+3;
        case 2
            %the operation is subtraction
            t{n}.type=2;
            t{n}.ln=0;
        case 3
            %the variable is x2
            t{n}.type=2;
            t{n}.ln=0;
        case 4
            %the digit is 2
            t{n}.type=2;
            t{n}.ln=0;
    end
case 3
    switch (t{n}.ev)
        case 1
            for pr=nn:-1:n+1,
               t{pr+2}.ev=t{pr}.ev;
               t{pr+2}.ln=t{pr}.ln;
               t{pr+2}.modder=t{pr}.modder;
               t{pr+2}.mb=t{pr}.mb;
               t{pr+2}.branch=t{pr}.branch;
            end
            t{n}.ln=2;
            t{n+1}.ev=4;
            t{n+1}.mb=n;
            t{n+1}.ln=-1;
            t{n+1}.modder=nd;
            t{n+1}.branch=2;
            t{n+2}.ev=3;
            t{n+2}.mb=n;
            t{n+2}.ln=-1;
            t{n+2}.modder=nv;
            t{n+2}.branch=1;
            nn=nn+2;
        case 2
            %operation is multiplication
```

```
                    t{n}.type=3;
                    t{n}.ln=0;
                case 4
                    %digit is 3
                    t{n}.type=3;
                    t{n}.ln=0;
            end
        case 4
            switch (t{n}.ev)
                case 1
                    for pr=nn:-1:n+1,
                       t{pr+2}.ev=t{pr}.ev;
                       t{pr+2}.ln=t{pr}.ln;
                       t{pr+2}.modder=t{pr}.modder;
                       t{pr+2}.mb=t{pr}.mb;
                       t{pr+2}.branch=t{pr}.branch;
                    end
                    t{n}.ln=2;
                    t{n+1}.ev=1;
                    t{n+1}.mb=n;
                    t{n+1}.ln=-1;
                    t{n+1}.modder=nnt;
                    t{n+1}.branch=2;
                    t{n+2}.ev=4;
                    t{n+2}.mb=n;
                    t{n+2}.ln=-1;
                    t{n+2}.modder=nd;
                    t{n+2}.branch=1;
                    nn=nn+2;
                case 4
                    %digit is 4
                    t{n}.type=4;
                    t{n}.ln=0;
            end
    end
    %forward pointer (don't need pointer for starting nonterminal)
    if n>1,
        t{t{n}.mb}.l(t{n}.branch)=n;
    end
    %move onto next leaf
    n=n+1;
    %update the current modder, which corresponds to specific leaf
    modder=t{n}.modder;
    %move to next number in gene
    counter = counter + 1;
end
%If tree not completed return invalid
if nn > n || (t{n}.ev ~=-2 && t{n}.ln == -1)
    clear t
    t{1}.ev = -10;
    t{1}.ln = -10;
end
rs = t;
return
```

```
%clean up the work space
clear all; home;
left = 1; %to verify which direction ran
nnt =   4;  % number of nonterminals
no =    3;  % number of operations
nv = 2; %number of variables
nd= 4; %number of digits
mst =   10;  % number of loops over the gene DNA
maxterminals = 4;

%   t is the function tree
%   gene is a DNA expression
%   cf are the compiled results -- i.e. groups of genes that result in the
%       same math expression

syms x1 x2 y0 y1 y2 y3
ind = 0;
%create the genes and empty array (ex) of strings
for m0 = 1:maxterminals,
   for m1 = 1:maxterminals,
       for m2 = 1:maxterminals,
           for m3 = 1:maxterminals,
               for m4 = 1:maxterminals,
                   for m5 = 1:maxterminals,
                       for m6 = 1:maxterminals,
                           for m7 = 1:maxterminals,
                               for m8 = 1:maxterminals,
                                   ind = ind + 1;
                                   gene1{ind} = [m0 m1 m2 m3 m4 m5 m6 m7 m8];
                                   ex1{ind} = ' ';
                               end
                           end
                       end
                   end
               end
           end
       end
    end
end
gl = length(gene1{1});
genospace = maxterminals^gl;
ccoff= zeros(1, genospace);
counters = 0;
%picks out 100000 random genes which are then run through the normal
%proceedings.
while counters < 100000
    r = round(1 + (genospace-1)*rand());
    if ccoff(r) == 0
        counters = counters+1;
        gene2{counters} = gene1{r};
    end
```

```
end
clear gene1

parfor ind = 1:100000,
    %to run in parallel
    ex1{ind} = double(char(toeval(build_strpolyparallel(1,...
        build_treeppLeftinvalid(gene2{ind}, mst, nnt, no, nv, nd)))));
end
clear t
clear gene2
coff = zeros(1, 100000);
i = 0;
for m = 1:100000
    if coff(m) == 0
        coff(m) = 1;
        %i gives the size of our phenotypic search space
        i = i + 1;
        %j gives the degeneracy of the expression
        j = 1;
        %cf{i}.ex save the expression
        cf{i}.ex = ex1{m};
        %compare the rest of the expressions
        for mm = m+1:100000
            if length(ex1{m}) == length(ex1{mm})
                if ex1{m} == ex1{mm}
                    j= j+1;
                    %changed to 1 so this expression won't be evaluated
                    %again
                    coff(mm) = 1;
                end
            end
        end
        %assign the expression a degenerate number
        cf{i}.n = j;
    end
end
%find the position and amount of invalid expressions
mmm = 0;
keepgoing = 0;
while keepgoing ==0 && mmm <= length(cf)
    mmm= mmm+1;
    if length(cf{mmm}.ex) == length(double(char('IV')))
        if cf{mmm}.ex == double(char('IV'))
            invalid = cf{mmm}.n;
            position = mmm;
            keepgoing = 1;
        end
    end
end
%remove invalid expressions from data
cf(position) = [];
```

```
function rs = toeval(expression)
syms x1 x2 NA real
%switch symbols to digits
y0=1;
y1=2;
y2=3;
y3=4;
%evaluate with digits substituted
    temp = eval(expression);
    %checks if you have division by zero or complex expressions
    if imag(temp)==0 && -isinf(temp)
        rs = temp;
    else
        %if complex or infinite NA is returned
        rs = NA;
    end
return
```

```
        Error using toeval (line 9)
        Not enough input arguments.
```

Published with MATLAB® R2014a

Appendix 2

```
syms x1 x2
genospace = length(ex1);
%convert string to matrix of integers
parfor ind = 1: genospace,
    ex2{ind} = double(char(ex1{ind}));
end
%initialize coff matrix
coff = zeros(1, genospace);
i = 0;
for m = 1:genospace
    if coff(m) == 0
        coff(m) = 1;
        %i gives the size of our phenotypic search space
        i = i + 1;
        %j gives the degeneracy of the expression
        j = 1;
        %cf{i}.ex save the expression
        cf{i}.ex = ex2{m};
        %compare the rest of the expressions
        for mm = m+1:genospace
            if length(ex2{m}) == length(ex2{mm})
                if ex2{m} == ex2{mm}
                    j= j+1;
                    %changed to 1 so this expression won't be evaluated
                    %again
                    coff(mm) = 1;
                end
            end
        end
        %assign the expression a degenerate number
        cf{i}.n = j;
    end
end
```

```
        Attempt to execute SCRIPT ex1 as a function:
        /Applications/MATLAB_R2014a.app/toolbox/robust/rctobsolete/mutools

        Error in doublecompare (line 2)
        genospace = length(ex1);
```

Published with MATLAB® R2014a

```
function output = characterize(expression)
syms x1 x2 NA real
output{1} = expression;
if ~isa(expression, 'double')
    x1 = x2;
    q = eval(expression);
    x1 = 1;
    r = eval(expression);
    x2 = 1;
    s = eval(expression);
else
    q= expression;
end
if isa(q, 'double')
    degree = 0;
else
    degree1 = length(sym2poly(q)) -1;
    if isa(r, 'double')
        degree2 = 0;
    else
        degree2 = length(sym2poly(r)) -1;
    end
    if isa(s, 'double')
        degree3 = 0;
    else
        degree3 = length(sym2poly(s)) -1;
    end
    degree = max([degree1 degree2 degree3]);
end
output{2} = degree;
if ~isa(expression, 'double')
    for index = 3: 2+(degree + 1)*(degree + 2)/2;
        x1 = index +7;
        x2 = index - 1;
        point = eval(expression);
        output{index} = point;
    end
else
    output{3} = expression;
end
return
```

```
        Error using characterize (line 3)
        Not enough input arguments.
```

Published with MATLAB® R2014a

```
% compile the list of similar functions and places them in cf
coff = zeros(1,gl^maxterminals);
i = 0;
for m = 1:gl^maxterminals,
    if (coff(m) == 0)
        coff(m) = 1;
        i = i+1;
        j = 1;
        cf{i}.gene{j} = gene{m};
        cf{i}.ex = ex{m};
        for mm = m+1:gl^maxterminals,
            if (strcmp(char(ex{m}),char(ex{mm})))
                j= j+1;
                coff(mm) = 1;
                cf{i}.gene{j} = gene{mm};
            end
        end
        cf{i}.n = j;
        cf{i}.p = j/(gl^maxterminals);
    end
end

for m = 1:i,
    disp(cf{m})
    dn(m)= cf{m}.n;
end
```

```
        Undefined function or variable 'gl'.

        Error in originalCompare (line 2)
        coff = zeros(1,gl^maxterminals);
```

Published with MATLAB® R2014a

```
splitby = genospace/2;
%splits phenospace into smaller lists
for ind1 = 1:splitby,
   for ind2 = (ind1-1)*genospace/splitby+1:ind1*genospace/splitby
        ex2{ind1}{mod(ind2, genospace/splitby)+1} = ex1{ind2};
   end
end
%runs grouping on smaller lists
parfor ghd=1:splitby
   biggroup{ghd}=grouping(ex2{ghd})
end
clear ex2
for var = 1: log2(genospace)-1
    bgind2 = 0;
    %forms a bigger group with 2 consecutive lists
    for bgind1 = 1:2:genospace/2^var -1
        bgind2 = bgind2 +1;
        ex2{bgind2} = cat(2, biggroup{bgind1}, biggroup{bgind1 +1});
        %allows us to compare only phenotypes which haven't been compared
        %yet
        ex3{bgind2}.m = length(biggroup{bgind1});
        ex3{bgind2}.n = length(biggroup{bgind1 +1});
    end
    clear biggroup
    %this runs grouping on newly formed groups
    parfor ghd=1:length(ex2)
       biggroup{ghd}=grouping2(ex2{ghd}, ex3{ghd}.m, ex3{ghd}.n)
    end

    clear ex2
    clear ex3
end

for m = 1:length(biggroup{1}),
    disp(biggroup{1}{m}.ex)
    disp(biggroup{1}{m})
   %dn(m)= cf{m}.n;
end
```

```
        Undefined function or variable 'genospace'.

        Error in parallelCompare (line 1)
        splitby = genospace/2;
```

Published with MATLAB® R2014a

```
average = length(coff)/length(cf)


total =0;
for mmm = 1: length(cf),
    total = total + (cf{mmm}.n - average)^2;
end
stdev = sqrt(total / length(cf))


tots = 0;

for mm = 1:length(cf),
    tots = cf{mm}.n/length(coff) * log2(cf{mm}.n/length(coff))+tots;
end
entropy = -1*tots
```

```
        Undefined function or variable 'coff'.

        Error in stats (line 1)
        average = length(coff)/length(cf)
```

Published with MATLAB® R2014a

Appendix 3

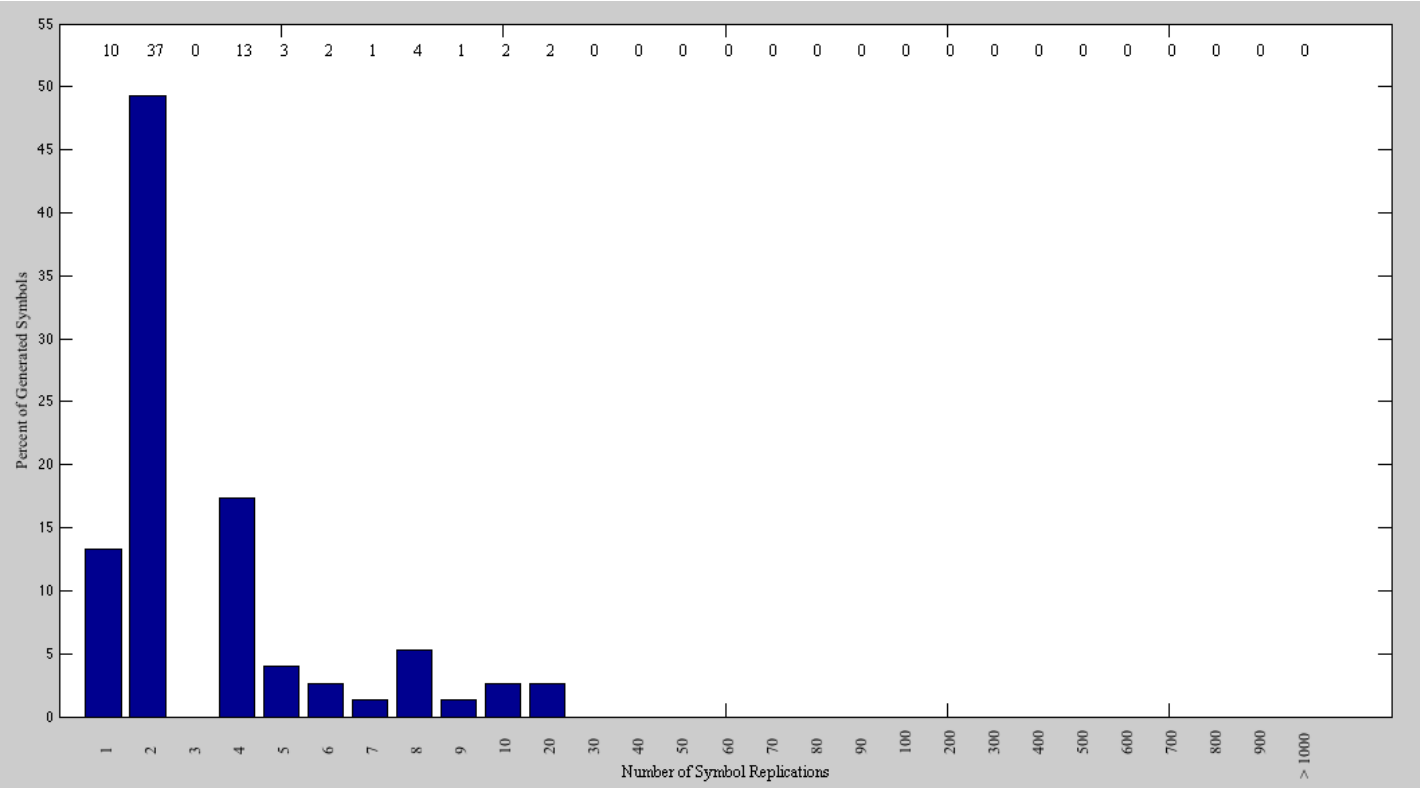
Left or Right	Gene Length	Wrapping	Mean	Standard Deviation	Entropy	Phenotypes	Genotypes	Percent
L	4	1	3.41	2.68	5.69	75	256	29.30
L	4	2	3.46	4.58	5.53	74	256	28.91
L	4	3	3.24	4.21	5.69	79	256	30.86
L	4	4	3.20	4.18	5.71	80	256	31.25
L	4	6	3.20	4.18	5.71	80	256	31.25
L	4	7	3.20	4.18	5.71	80	256	31.25
L	4	8	3.20	4.18	5.71	80	256	31.25
L	4	9	3.20	4.18	5.71	80	256	31.25
L	4	10	3.20	4.18	5.71	80	256	31.25
R	4	1	2.51	2.15	6.26	102	256	39.84
R	4	2	2.44	4.08	5.83	105	256	41.02
R	4	3	2.12	3.76	6.08	121	256	47.27
R	4	4	2.15	3.78	6.07	119	256	46.48
R	4	5	2.13	3.78	6.06	120	256	46.88
R	4	6	2.21	3.84	6.03	116	256	45.31
R	4	7	2.08	3.72	6.10	123	256	48.05
R	4	8	2.19	3.82	6.04	117	256	45.70
R	4	9	2.12	3.76	6.08	121	256	47.27
R	4	10	2.15	3.78	6.07	119	256	46.48
L	5	1	3.52	6.18	7.21	291	1024	28.42
L	5	2	2.69	7.36	7.11	380	1024	37.11
L	5	3	2.22	4.62	7.83	461	1024	45.02
L	5	4	2.39	6.95	7.25	429	1024	41.89
L	5	5	2.14	4.42	7.92	479	1024	46.78
L	5	6	2.33	6.85	7.27	439	1024	42.87
L	5	7	2.08	4.36	7.96	493	1024	48.14
L	5	8	2.40	6.95	7.24	427	1024	41.70
L	5	9	2.08	4.35	7.96	493	1024	48.14
L	5	10	2.38	6.92	7.25	431	1024	42.09
R	5	1	2.73	5.22	7.58	375	1024	36.62
R	5	2	2.68	7.02	7.18	382	1024	37.30
R	5	3	2.22	4.73	7.81	461	1024	45.02
R	5	4	2.56	7.04	7.17	400	1024	39.06
R	5	5	2.19	4.65	7.84	467	1024	45.61
R	5	6	2.56	7.04	7.17	400	1024	39.06
R	5	7	2.19	4.65	7.84	468	1024	45.70
R	5	8	2.55	7.03	7.17	401	1024	39.16
R	5	9	2.19	4.65	7.84	468	1024	45.70
R	5	10	2.56	7.04	7.17	400	1024	39.06

L	6	1	6.95	17.38	7.94	589	4096	14.38
L	6	2	5.70	17.21	7.88	718	4096	17.53
L	6	3	4.72	14.05	8.39	867	4096	21.17
L	6	4	5.03	15.83	8.16	815	4096	19.90
L	6	5	4.68	14.04	8.40	875	4096	21.36
L	6	6	4.93	16.65	8.08	830	4096	20.26
L	6	7	4.60	13.85	8.44	891	4096	21.75
L	6	8	4.97	15.86	8.14	824	4096	20.12
L	6	9	4.60	13.82	8.45	891	4096	21.75
L	6	10	4.81	15.09	8.25	851	4096	20.78
R	6	1	5.07	14.75	8.19	808	4096	19.73
R	6	2	3.99	14.68	8.11	1026	4096	25.05
R	6	3	2.80	11.07	8.79	1463	4096	35.72
R	6	4	2.85	11.95	8.60	1435	4096	35.03
R	6	5	2.47	10.41	8.93	1655	4096	40.41
R	6	6	2.64	12.27	8.60	1553	4096	37.92
R	6	7	2.52	10.52	8.89	1625	4096	39.67
R	6	8	2.59	11.58	8.68	1583	4096	38.65
R	6	9	2.49	10.43	8.92	1647	4096	40.21
R	6	10	2.57	11.26	8.74	1595	4096	38.94
L	7	1	8.53	39.71	8.80	1920	16384	11.72
L	7	2	3.79	27.66	9.50	4319	16384	26.36
L	7	3	2.78	21.15	10.18	5898	16384	36.00
L	7	4	2.71	22.93	9.96	6043	16384	36.88
L	7	5	2.43	19.65	10.37	6730	16384	41.08
L	7	6	2.43	21.67	10.10	6741	16384	41.14
L	7	7	2.34	19.27	10.40	7005	16384	42.76
L	7	8	2.35	21.30	10.13	6974	16384	42.57
L	7	9	2.29	19.06	10.43	7160	16384	43.70
L	7	10	2.36	21.36	10.12	6933	16384	42.32
R	7	1	6.05	32.13	9.28	2708	16384	16.53
R	7	2	3.18	24.20	9.87	5154	16384	31.46
R	7	3	2.33	19.33	10.46	7040	16384	42.97
R	7	4	2.15	19.78	10.36	7630	16384	46.57
R	7	5	2.02	17.98	10.62	8114	16384	49.52
R	7	6	2.02	19.17	10.44	8123	16384	49.58
R	7	7	1.95	17.68	10.66	8386	16384	51.18
R	7	8	1.97	18.93	10.47	8322	16384	50.79
R	7	9	1.93	17.55	10.68	8507	16384	51.92
R	7	10	1.96	18.88	10.48	8376	16384	51.12

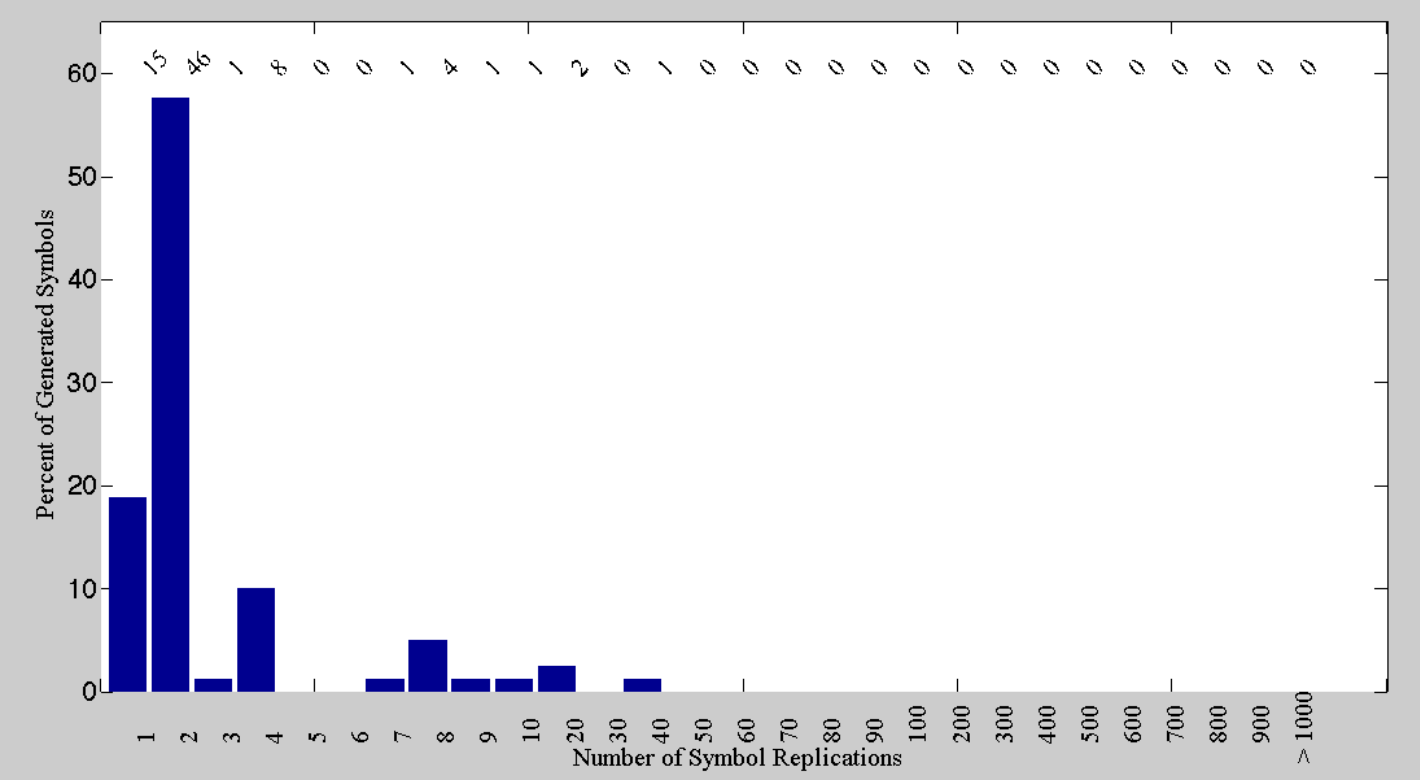
L	8	10	6.73	65.94	10.54	9743	65536	14.87
R	8	5	2.60	40.66	11.45	25195	65536	38.44
L	9	10	3.76	80.46	10.74	26609	100000	26.61
R	9	5	3.68	102.11	10.67	27177	100000	27.18
L	10	10	5.33	46.44	12.56	18756	100000	18.76
R	10	5	3.08	9.25	13.78	32490	100000	32.49

Appendix 4

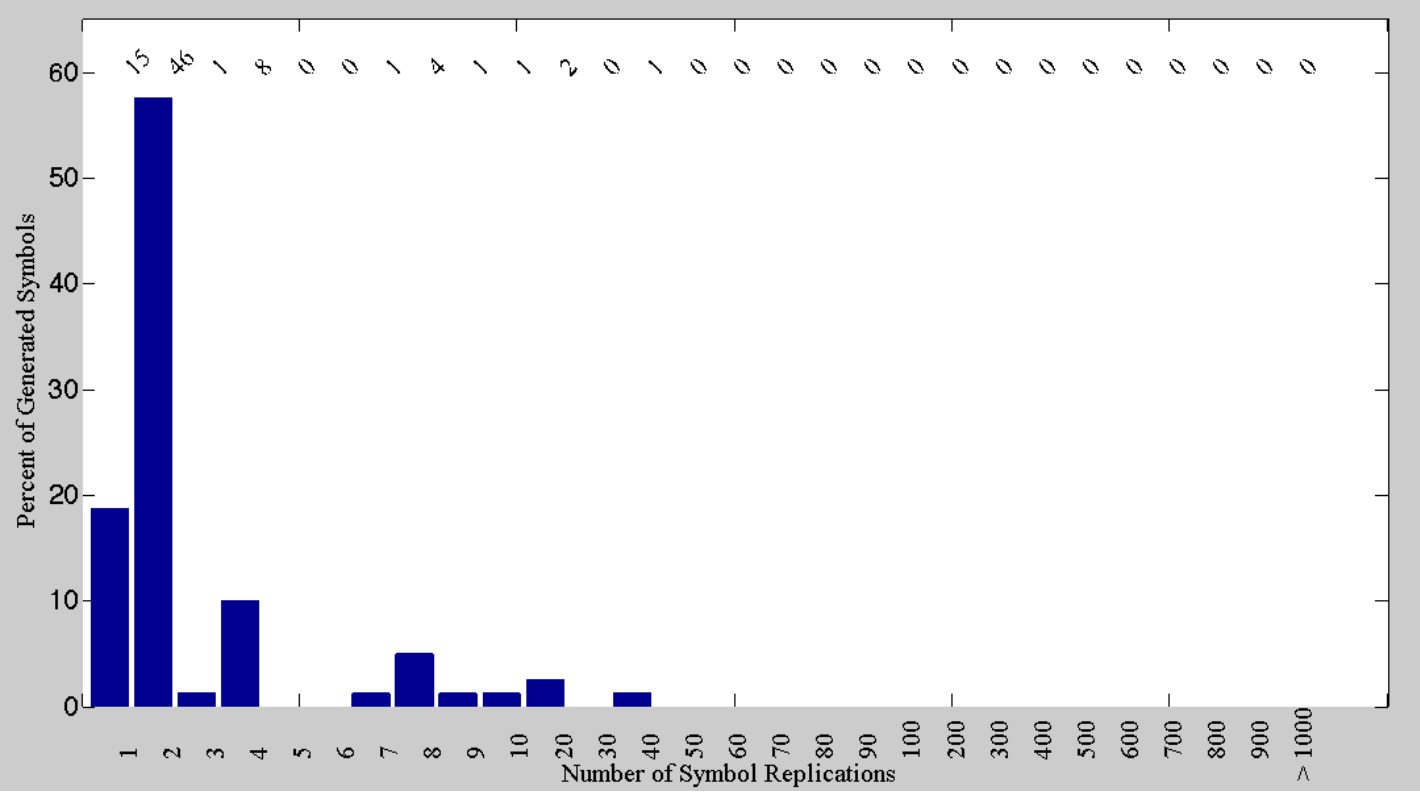
L41V



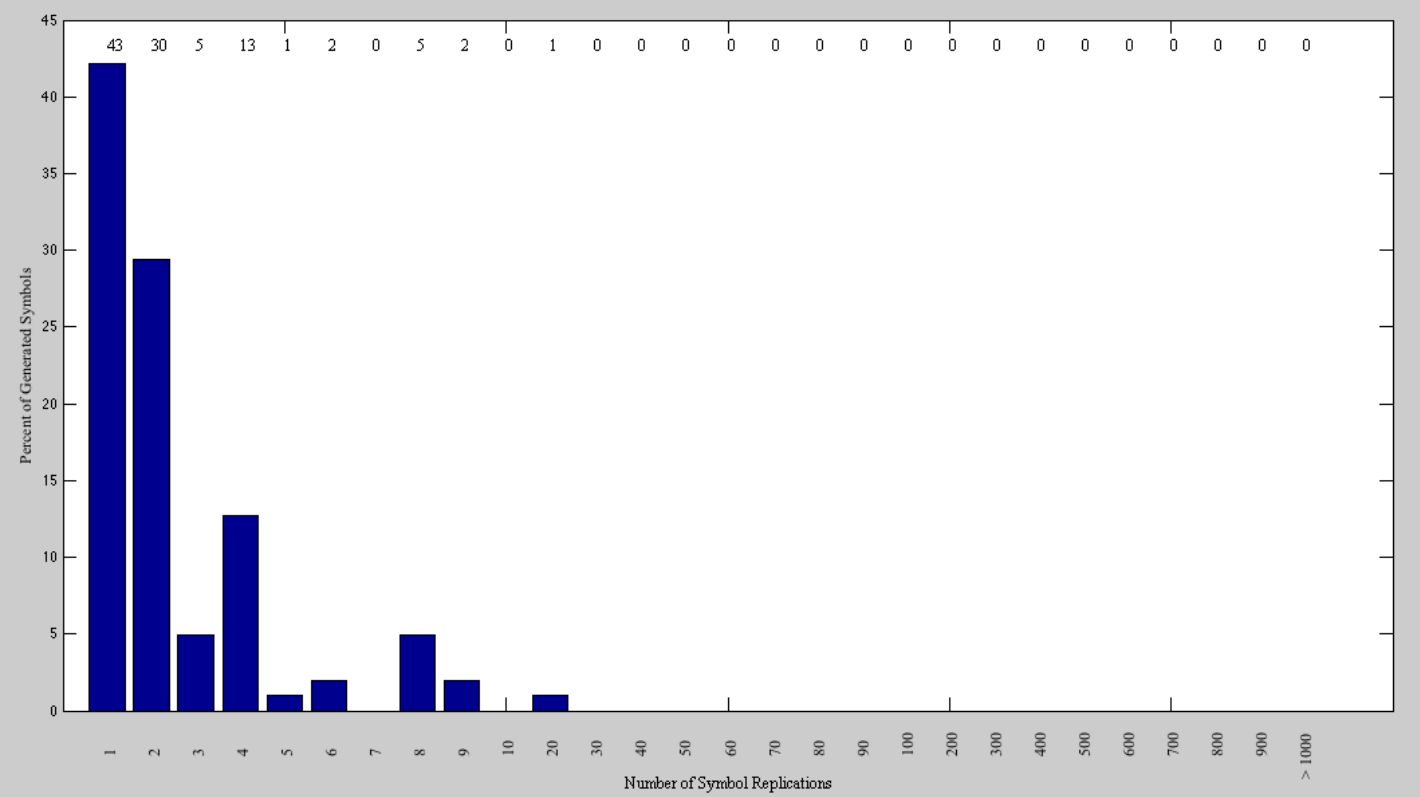
L45V



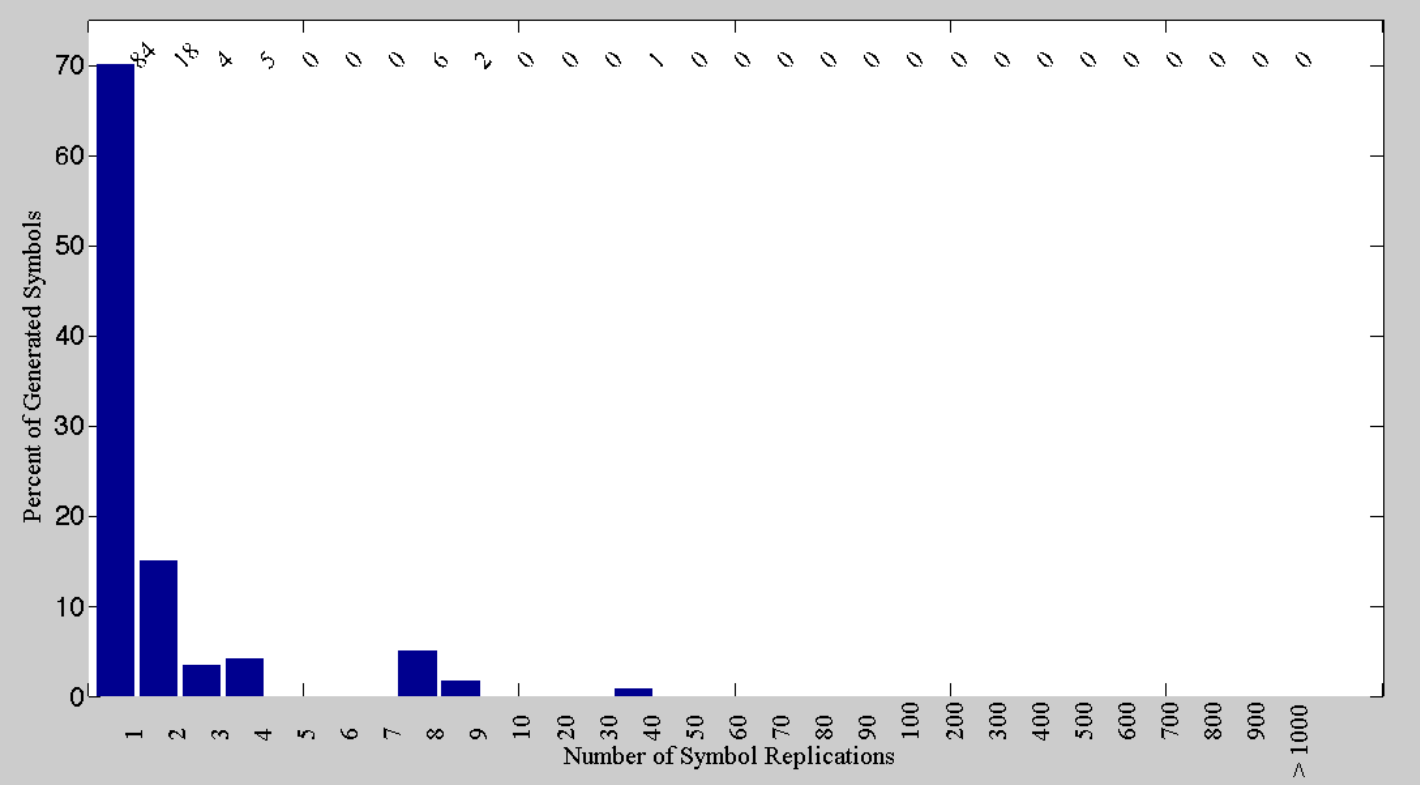
L410V


15 46 1 8 0 0 1 4 1 1 2 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Percent of Generated Symbols
0 10 20 30 40 50 60
1 2 3 4 5 6 7 8 9 10 20 30 40 50 60 70 80 90 100 200 300 400 500 600 700 800 900 > 1000
Number of Symbol Replications

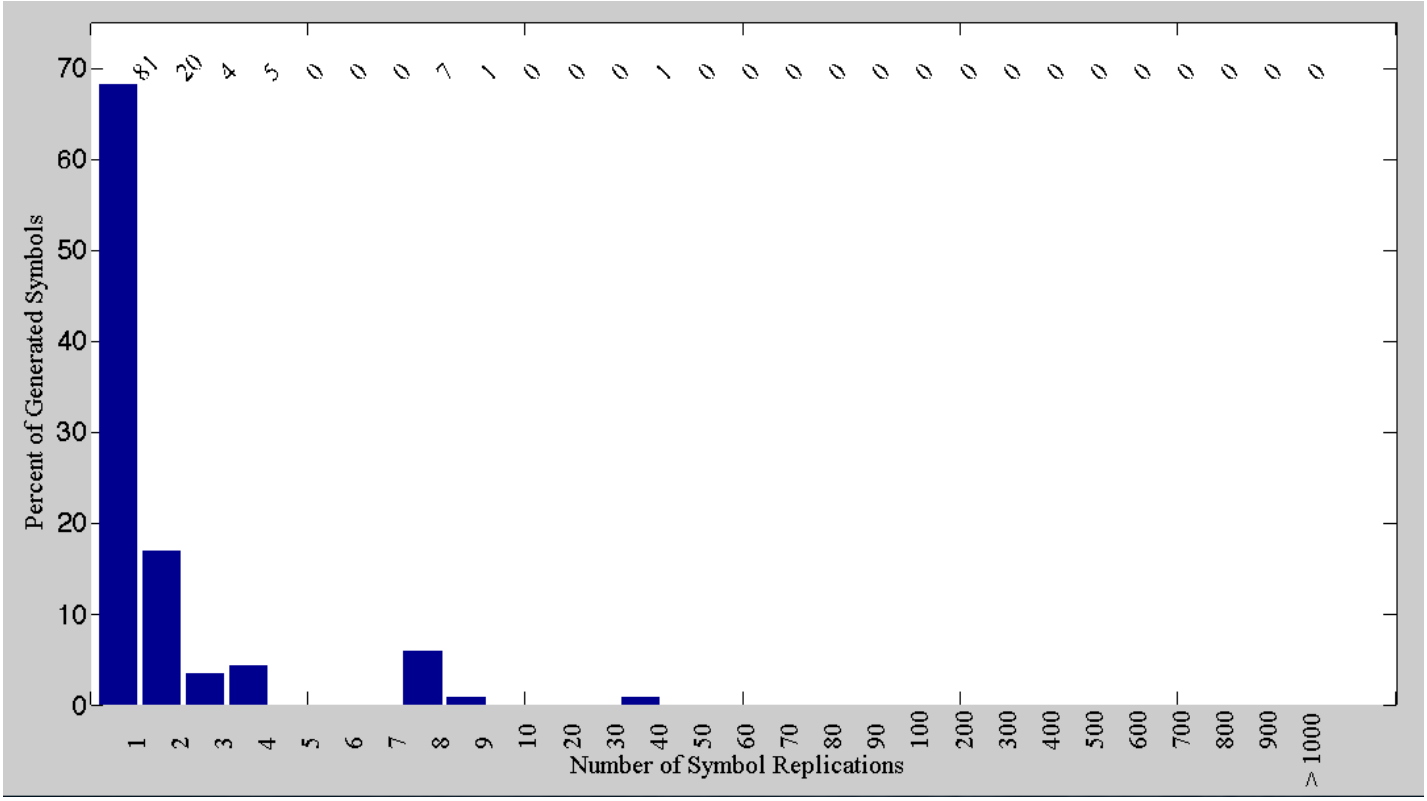
R41V


43 30 5 13 1 2 0 5 2 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Percent of Generated Symbols
0 5 10 15 20 25 30 35 40 45
1 2 3 4 5 6 7 8 9 10 20 30 40 50 60 70 80 90 100 200 300 400 500 600 700 800 900 > 1000
Number of Symbol Replications

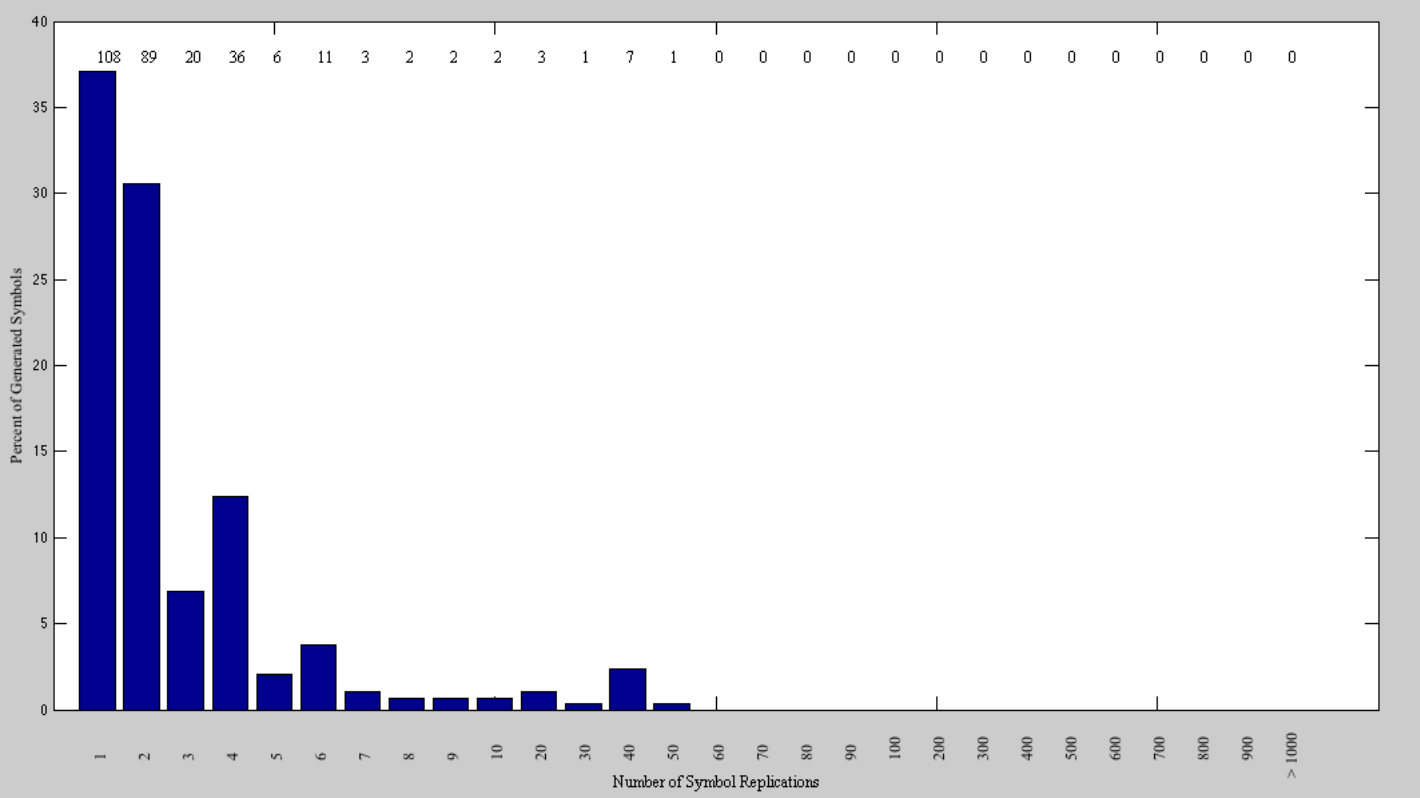
R45V


Percent of Generated Symbols
0
10
20
30
40
50
60
70
84
18
4
5
0
0
0
6
2
0
0
0
1
0
0
0
0
0
0
0
0
0
0
0
0
0
0
1
2
3
4
5
6
7
8
9
10
20
30
40
50
60
70
80
90
100
200
300
400
500
600
700
800
900
> 1000
Number of Symbol Replications

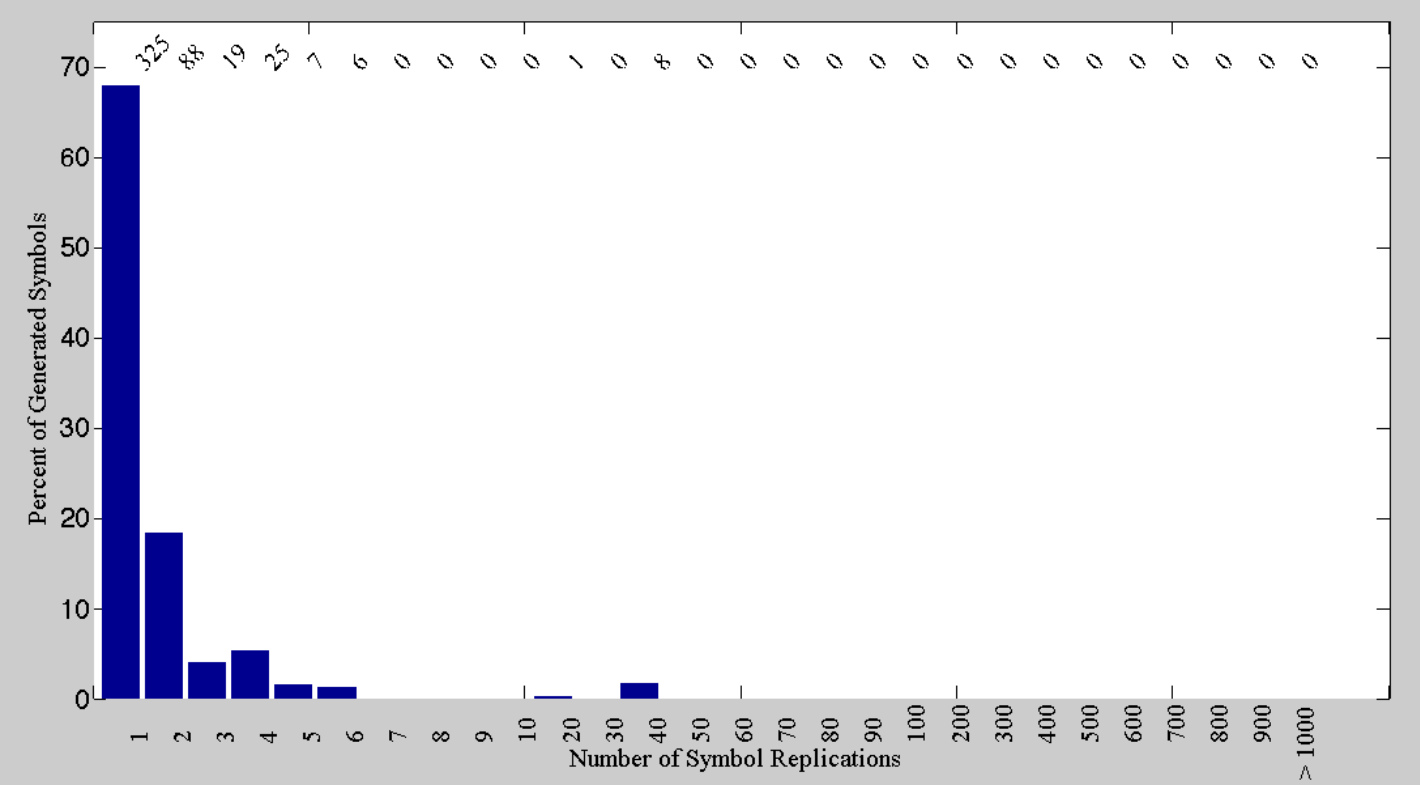
R410V


Percent of Generated Symbols
0
10
20
30
40
50
60
70
81
20
4
5
0
0
0
7
1
0
0
0
1
0
0
0
0
0
0
0
0
0
0
0
0
0
0
1
2
3
4
5
6
7
8
9
10
20
30
40
50
60
70
80
90
100
200
300
400
500
600
700
800
900
> 1000
Number of Symbol Replications

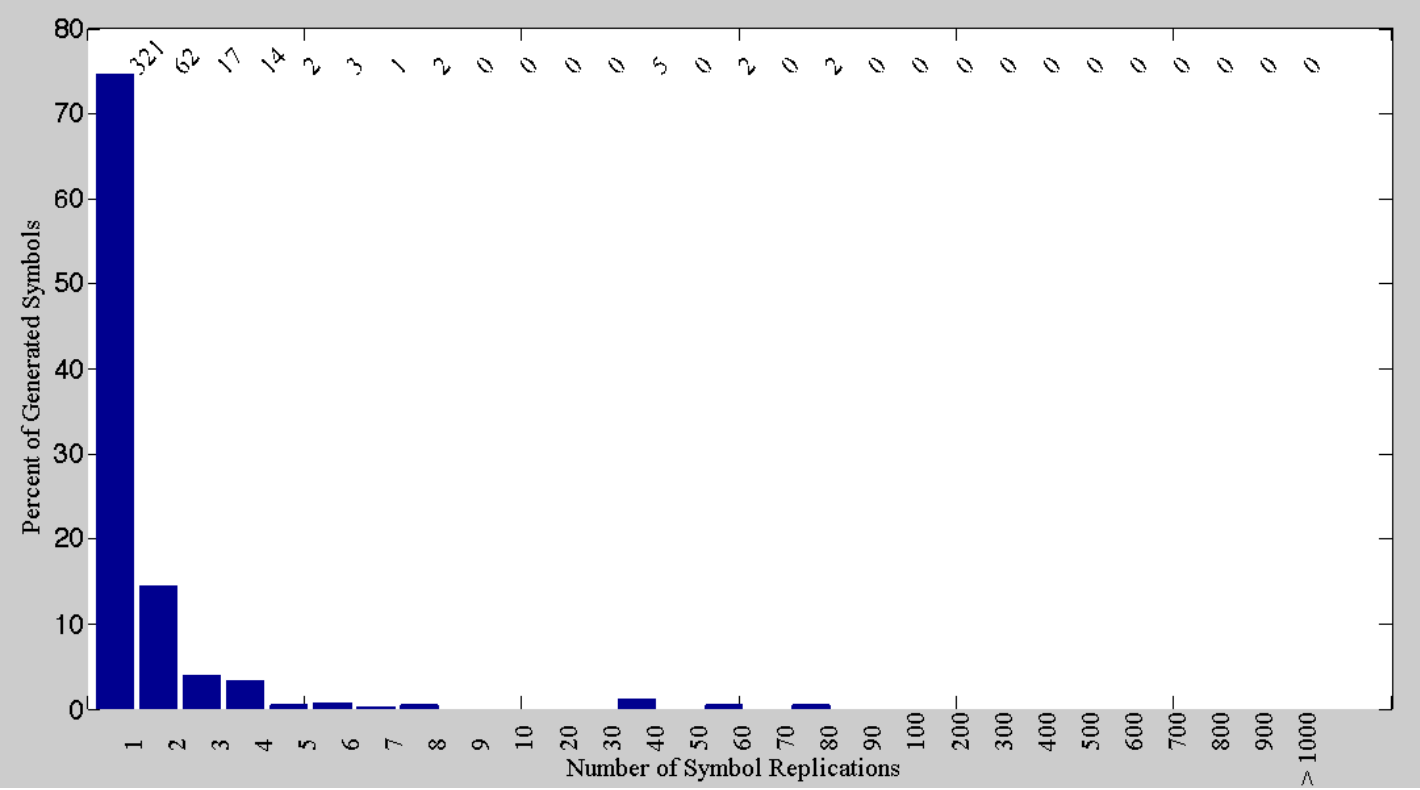
L51V


108 89 20 36 6 11 3 2 2 2 3 1 7 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Percent of Generated Symbols
0 5 10 15 20 25 30 35 40
1 2 3 4 5 6 7 8 9 10 20 30 40 50 60 70 80 90 100 200 300 400 500 600 700 800 900 > 1000
Number of Symbol Replications

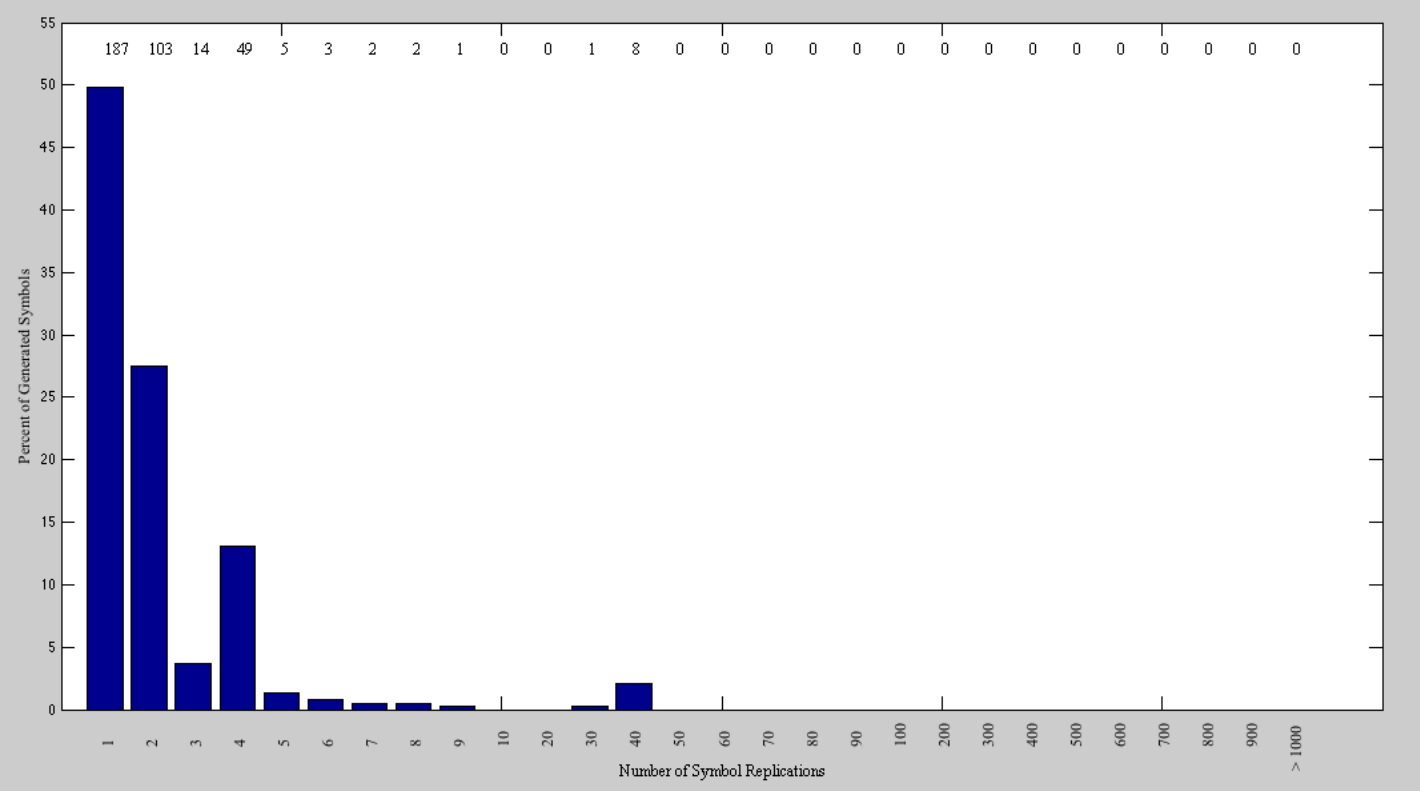
L55V


325 88 19 25 7 6 0 0 0 0 1 0 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Percent of Generated Symbols
0 10 20 30 40 50 60 70
1 2 3 4 5 6 7 8 9 10 20 30 40 50 60 70 80 90 100 200 300 400 500 600 700 800 900 > 1000
Number of Symbol Replications

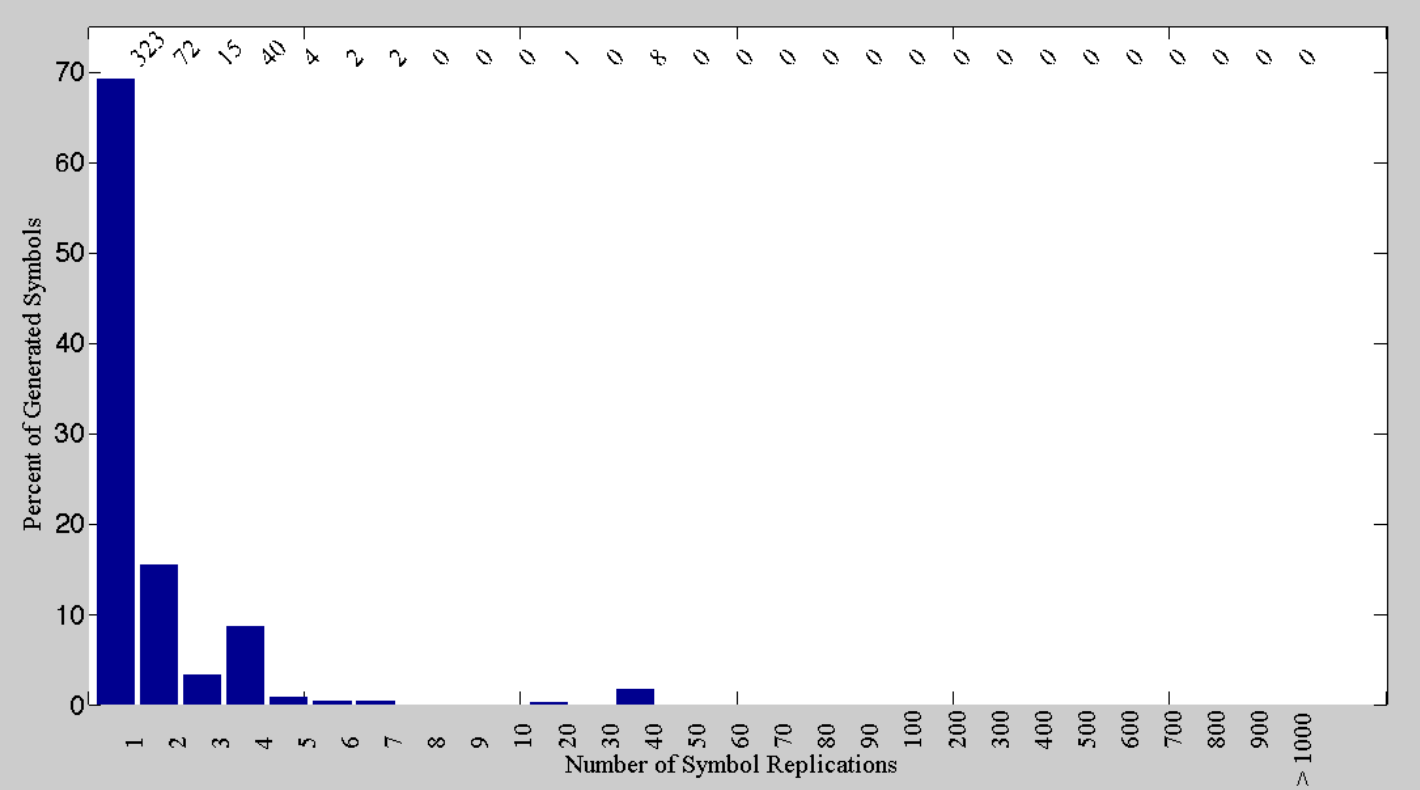
L510V


Percent of Generated Symbols
Number of Symbol Replications
321 62 17 14 2 3 1 2 0 0 0 0 5 0 2 0 2 0 0 0 0 0 0 0 0 0 0
1 2 3 4 5 6 7 8 9 10 20 30 40 50 60 70 80 90 100 200 300 400 500 600 700 800 900 > 1000

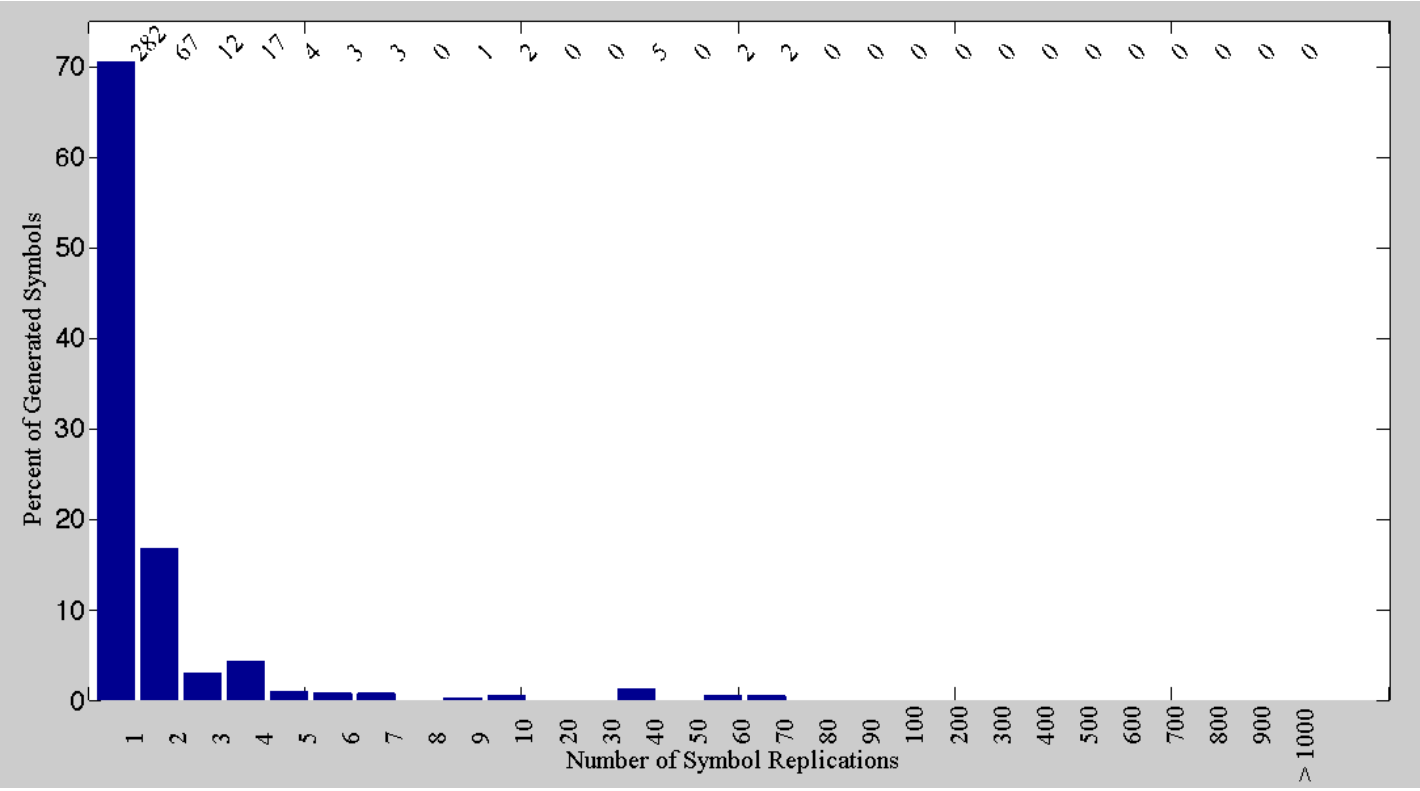
R51V


Percent of Generated Symbols
Number of Symbol Replications
187 103 14 49 5 3 2 2 1 0 0 1 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 2 3 4 5 6 7 8 9 10 20 30 40 50 60 70 80 90 100 200 300 400 500 600 700 800 900 > 1000

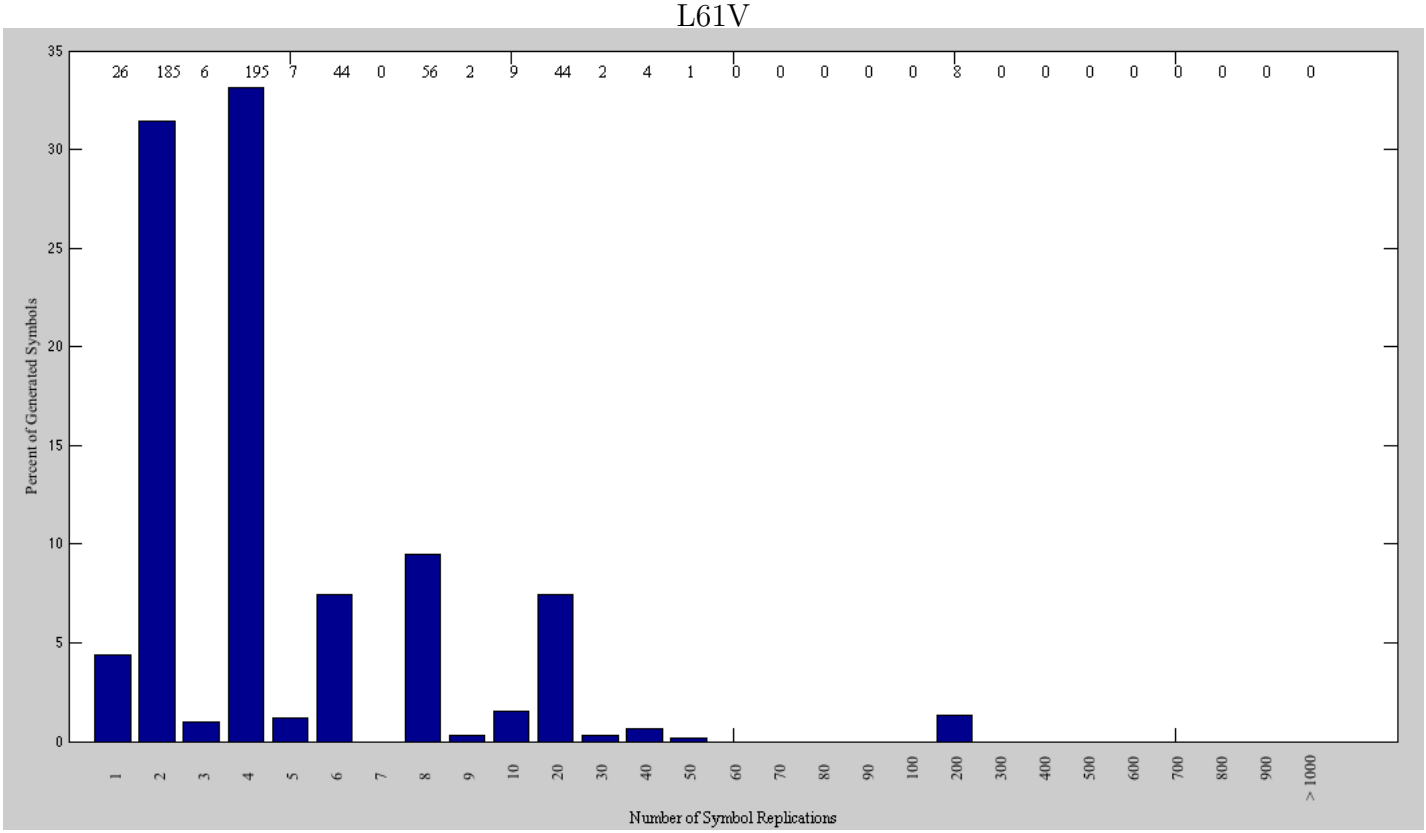
R55V


323
72
15
40
4
2
2
0
0
0
1
0
8
0
0
0
0
0
0
0
0
0
0
0
0
0
0
70
60
50
40
30
20
10
0
Percent of Generated Symbols
1
2
3
4
5
6
7
8
9
10
20
30
40
50
60
70
80
90
100
200
300
400
500
600
700
800
900
> 1000
Number of Symbol Replications

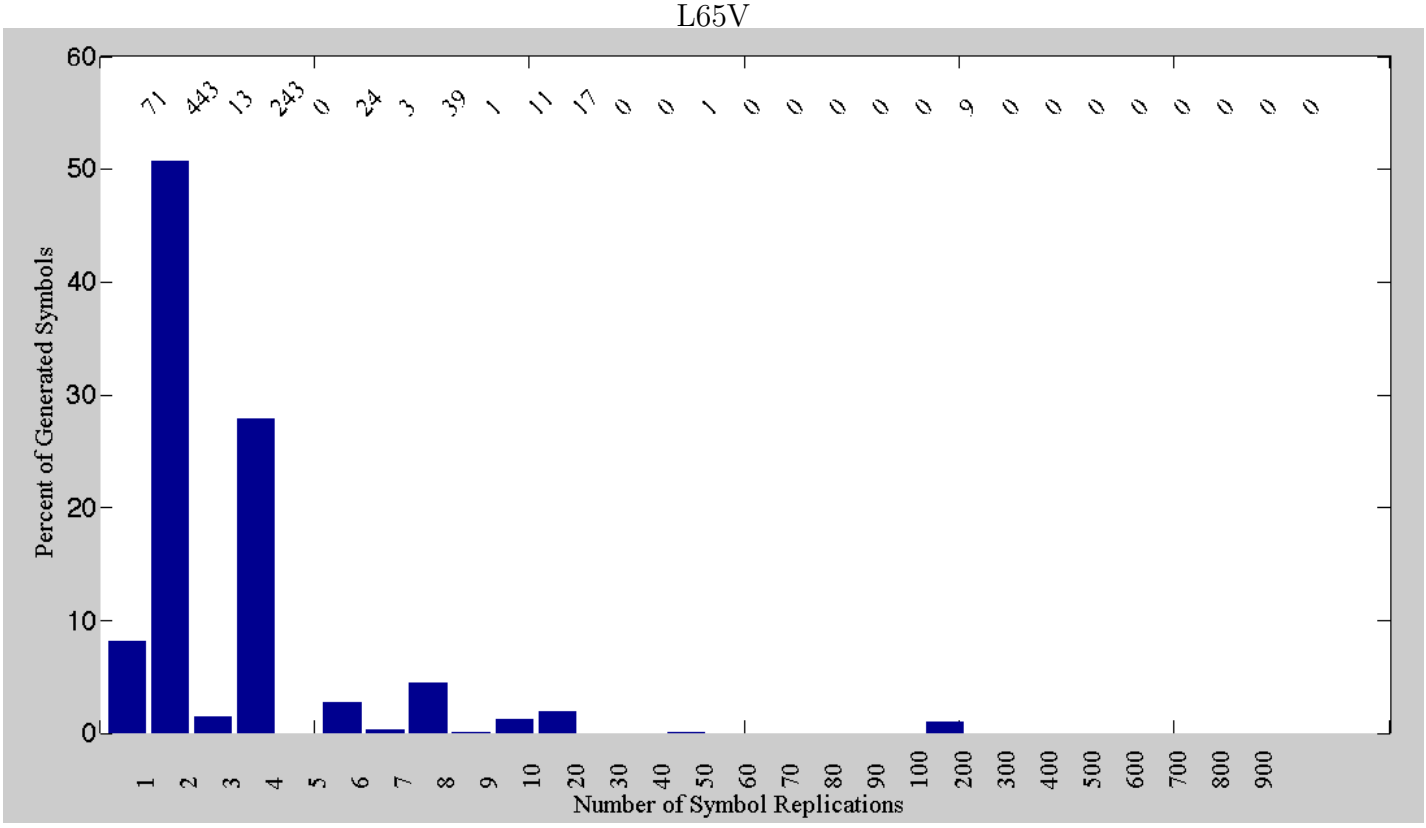
R510V


282
67
12
17
4
3
3
0
1
2
0
0
5
0
2
2
0
0
0
0
0
0
0
0
0
0
0
70
60
50
40
30
20
10
0
Percent of Generated Symbols
1
2
3
4
5
6
7
8
9
10
20
30
40
50
60
70
80
90
100
200
300
400
500
600
700
800
900
> 1000
Number of Symbol Replications

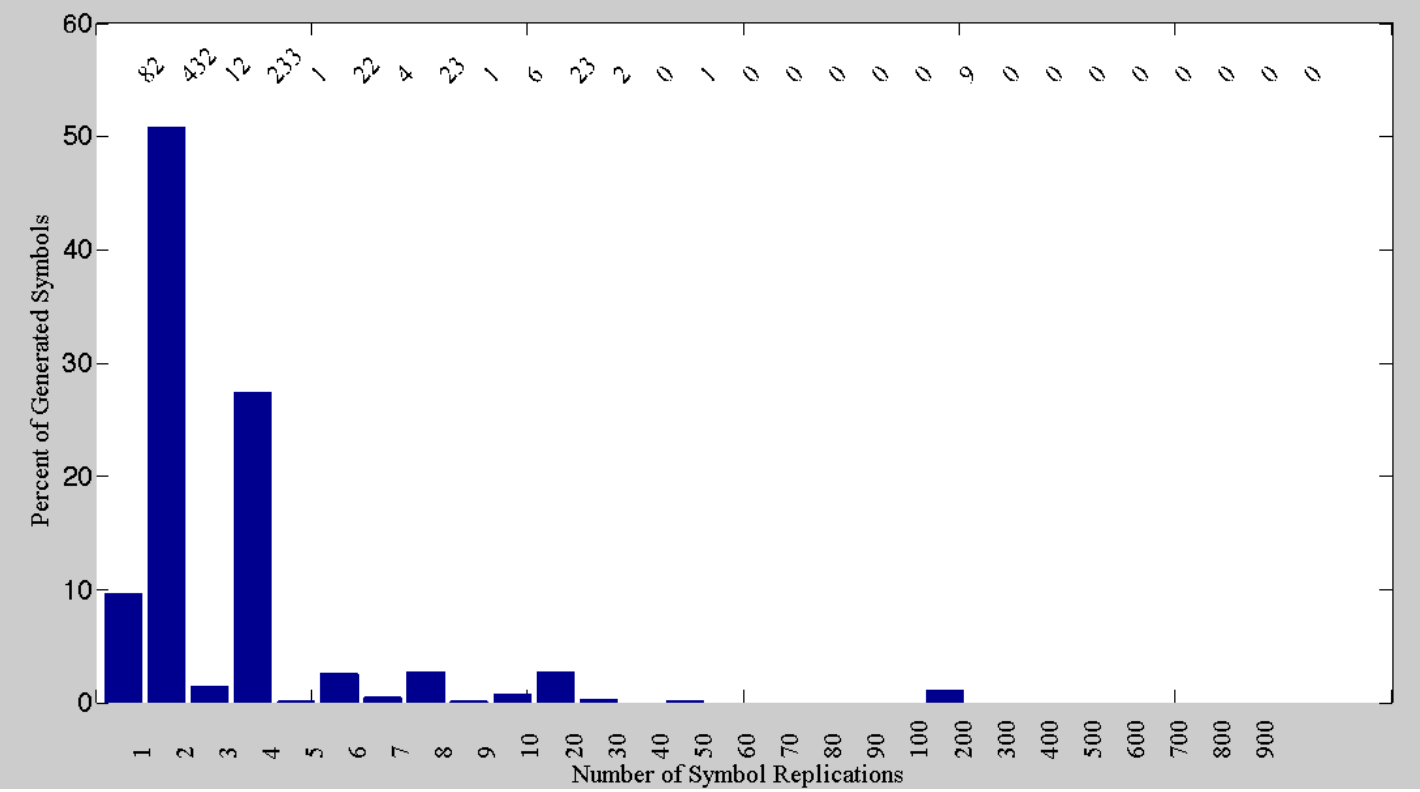
L61V


26 185 6 195 7 44 0 56 2 9 44 2 4 1 0 0 0 0 0 8 0 0 0 0 0 0 0 0
Percent of Generated Symbols
0 5 10 15 20 25 30 35
1 2 3 4 5 6 7 8 9 10 20 30 40 50 60 70 80 90 100 200 300 400 500 600 700 800 900 > 1000
Number of Symbol Replications

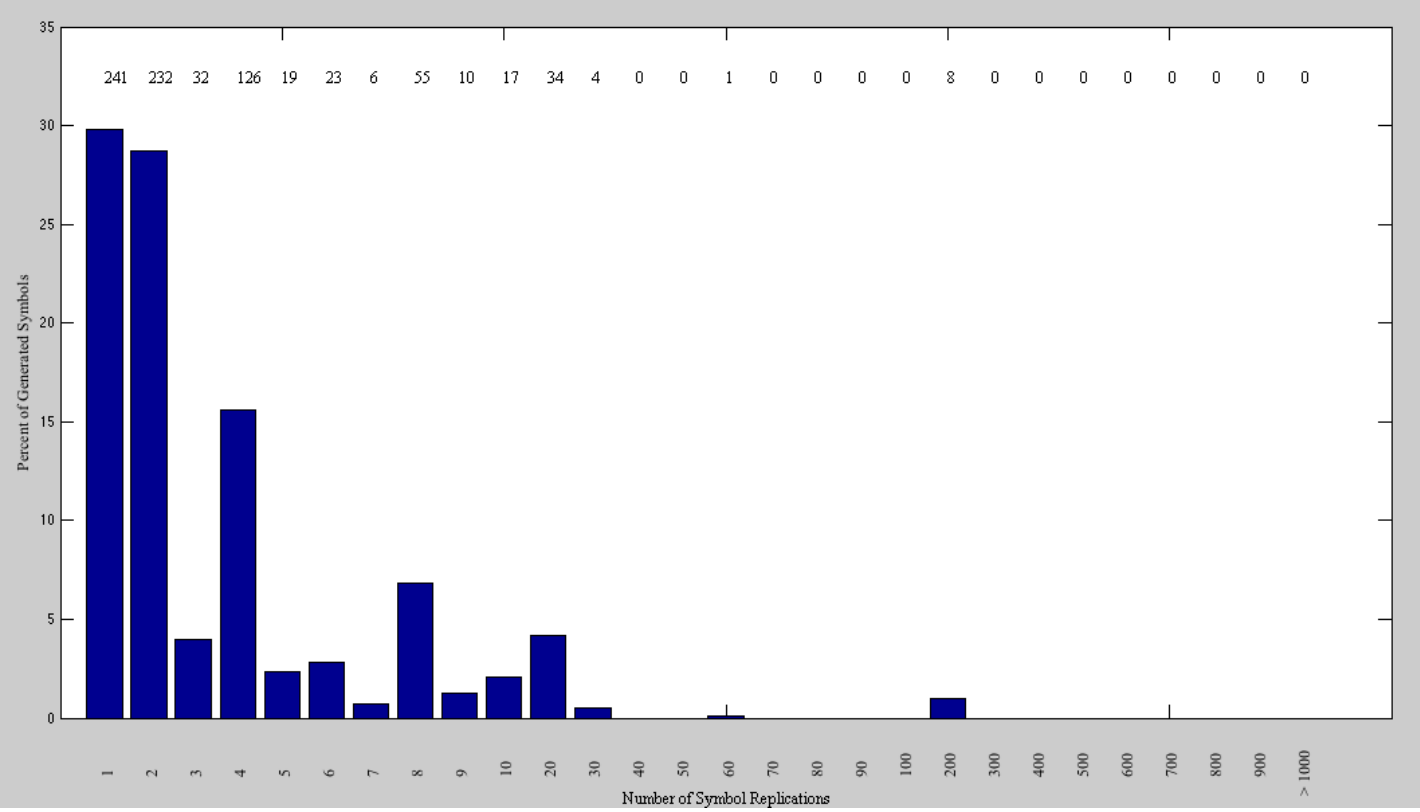
L65V


71 443 13 243 0 24 3 39 1 11 17 0 0 1 0 0 0 0 0 9 0 0 0 0 0 0 0 0
Percent of Generated Symbols
0 10 20 30 40 50 60
1 2 3 4 5 6 7 8 9 10 20 30 40 50 60 70 80 90 100 200 300 400 500 600 700 800 900
Number of Symbol Replications

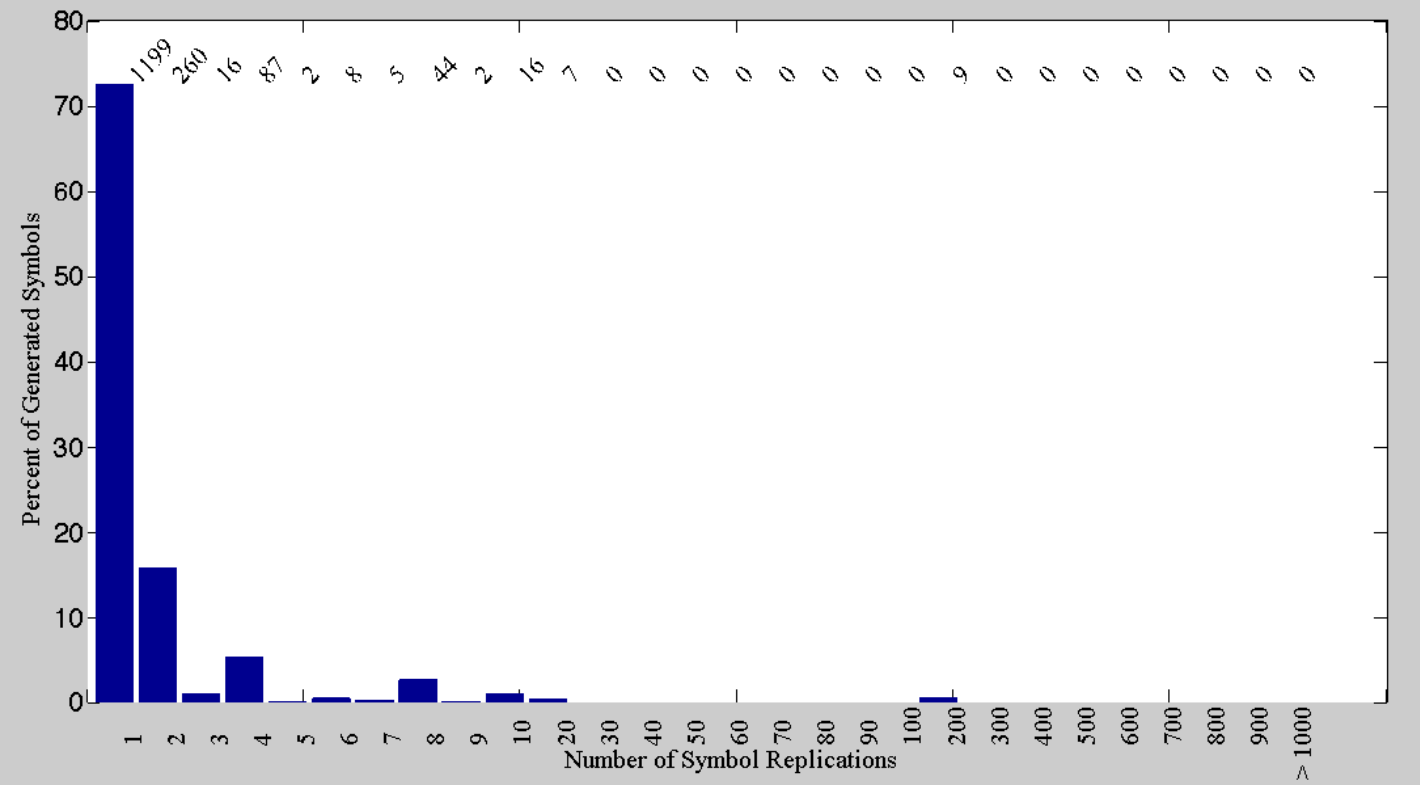
L610V


82 432 12 233 1 22 4 23 1 6 23 2 0 1 0 0 0 0 0 9 0 0 0 0 0 0 0 0 0
Percent of Generated Symbols
0 10 20 30 40 50 60
1 2 3 4 5 6 7 8 9 10 20 30 40 50 60 70 80 90 100 200 300 400 500 600 700 800 900
Number of Symbol Replications

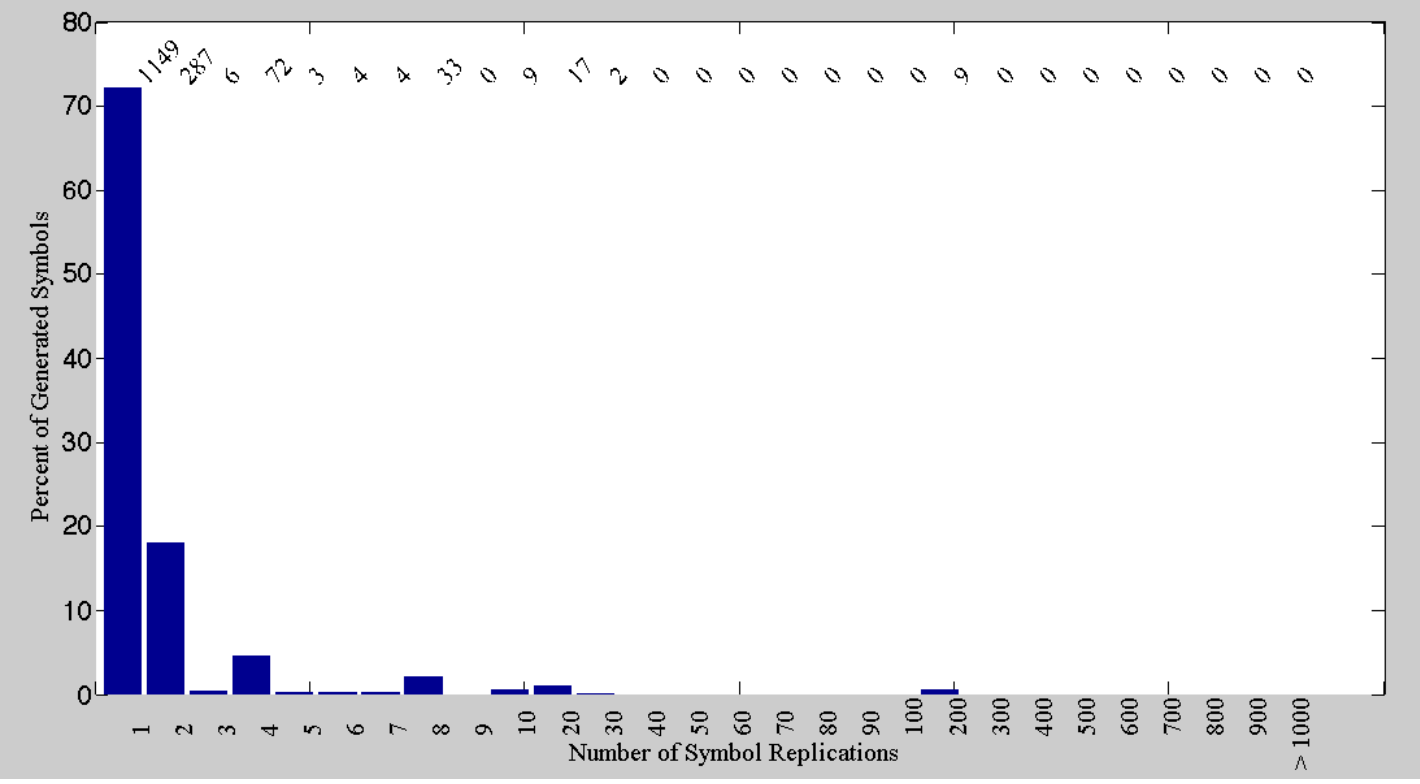
R61V


241 232 32 126 19 23 6 55 10 17 34 4 0 0 1 0 0 0 0 8 0 0 0 0 0 0 0 0 0
Percent of Generated Symbols
0 5 10 15 20 25 30 35
1 2 3 4 5 6 7 8 9 10 20 30 40 50 60 70 80 90 100 200 300 400 500 600 700 800 900 > 1000
Number of Symbol Replications

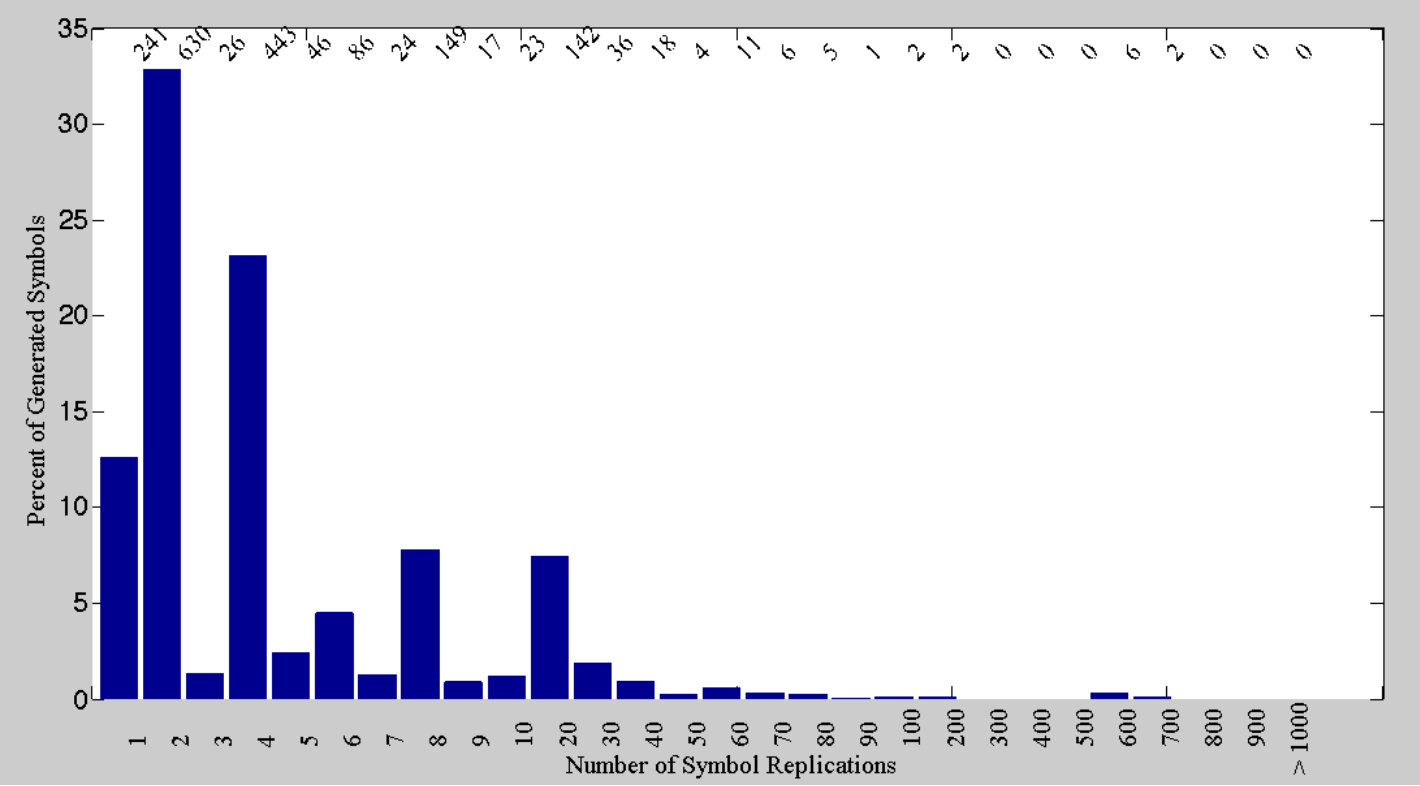
R65V


Percent of Generated Symbols
Number of Symbol Replications
1199
260
16
87
2
8
5
44
2
16
7
0
0
0
0
0
0
0
0
9
0
0
0
0
0
0
0
0
1
2
3
4
5
6
7
8
9
10
20
30
40
50
60
70
80
90
100
200
300
400
500
600
700
800
900
> 1000

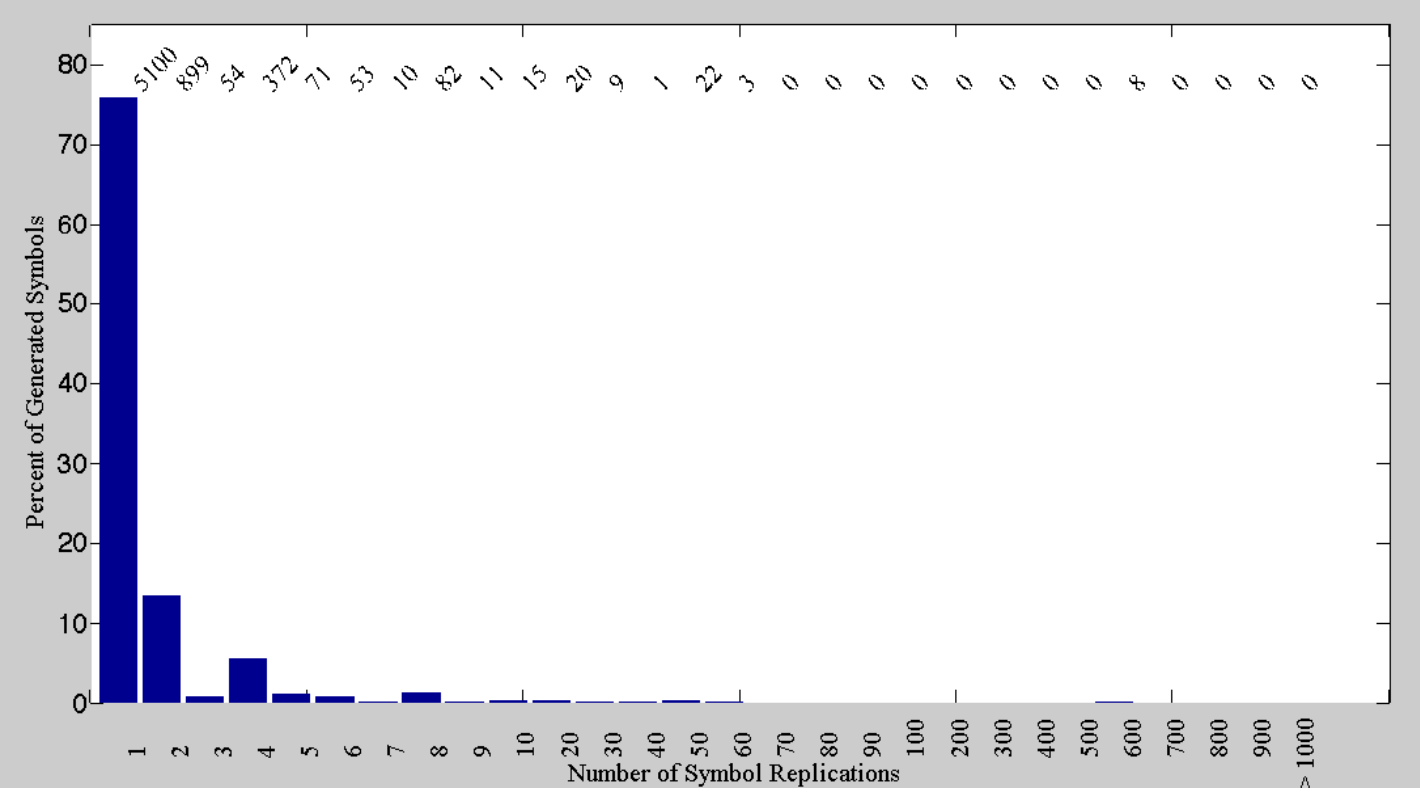
R610V


Percent of Generated Symbols
Number of Symbol Replications
1149
287
6
72
3
4
4
33
0
9
17
2
0
0
0
0
0
0
0
9
0
0
0
0
0
0
0
0
1
2
3
4
5
6
7
8
9
10
20
30
40
50
60
70
80
90
100
200
300
400
500
600
700
800
900
> 1000

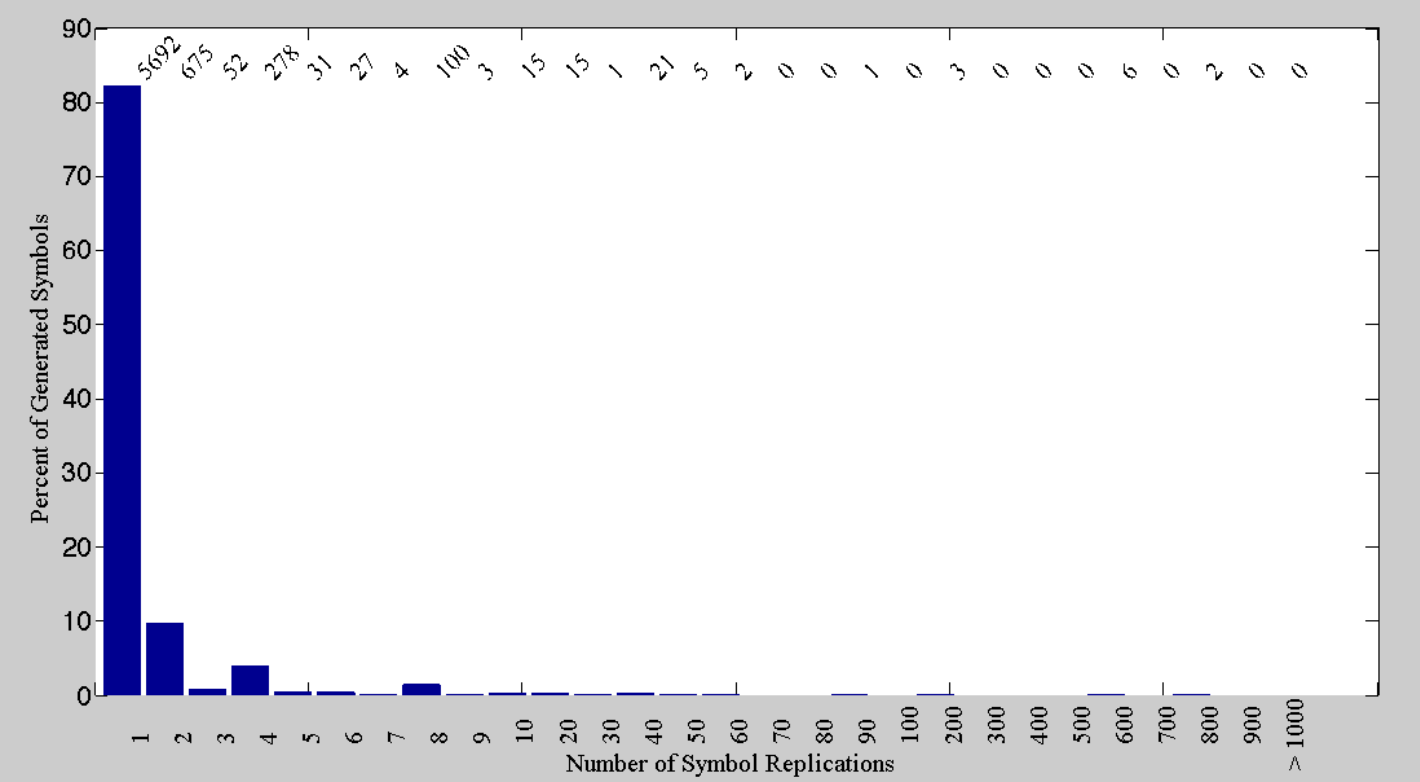
L71V


Percent of Generated Symbols
Number of Symbol Replications
241 630 26 443 46 86 24 149 17 23 142 36 18 4 11 6 5 1 2 2 0 0 0 6 2 0 0 0
1 2 3 4 5 6 7 8 9 10 20 30 40 50 60 70 80 90 100 200 300 400 500 600 700 800 900 > 1000

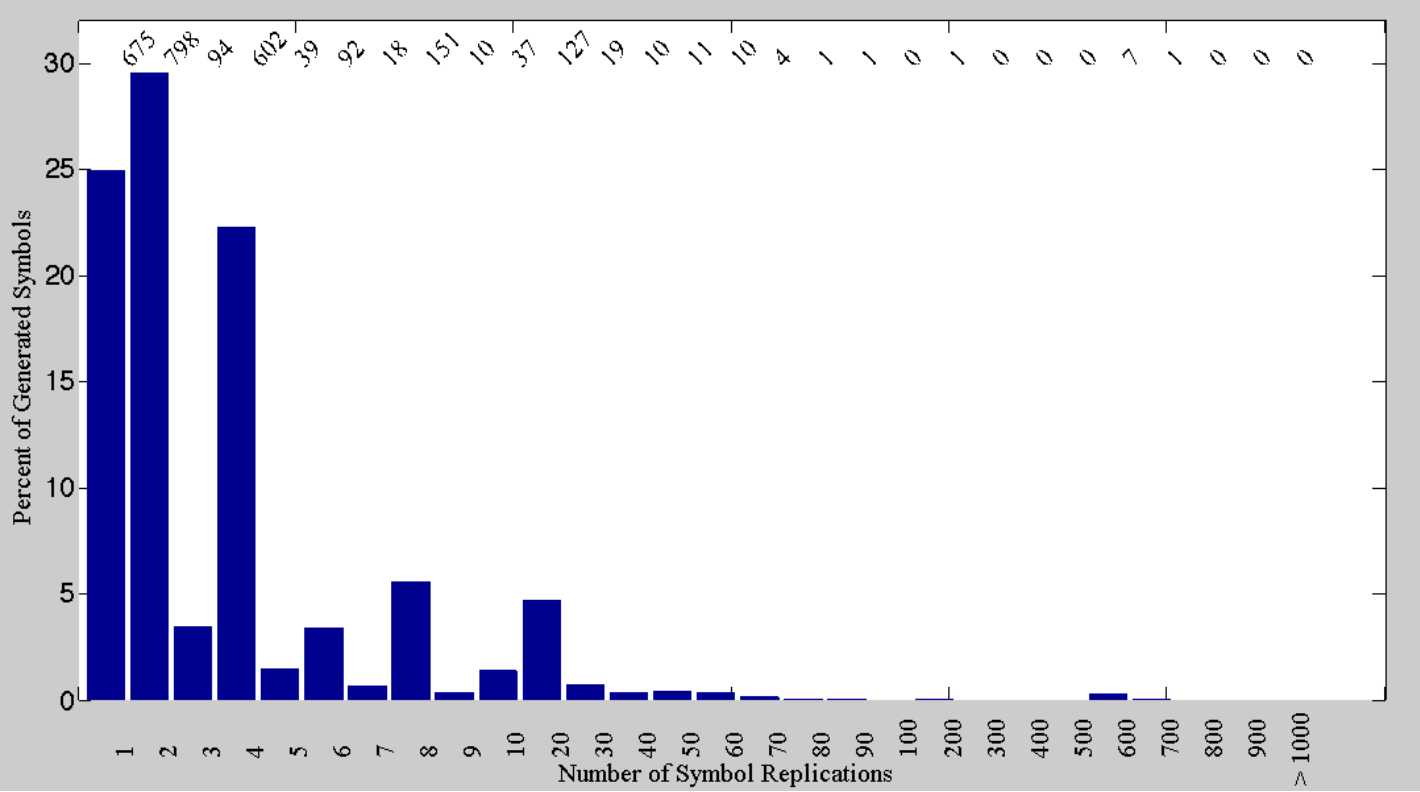
L75V


Percent of Generated Symbols
Number of Symbol Replications
5100 899 54 372 71 53 10 82 11 15 20 9 1 22 3 0 0 0 0 0 0 0 0 8 0 0 0 0
1 2 3 4 5 6 7 8 9 10 20 30 40 50 60 70 80 90 100 200 300 400 500 600 700 800 900 > 1000

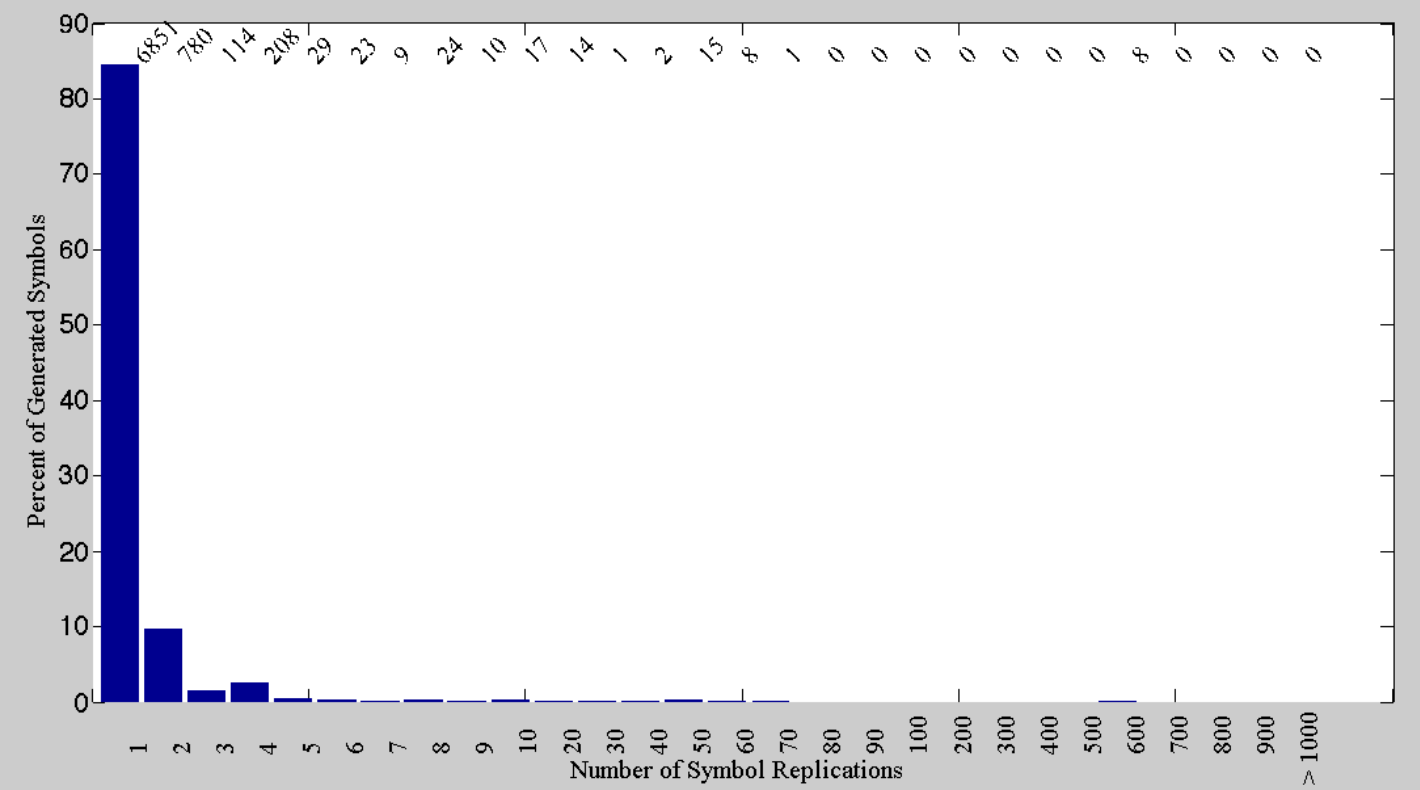
L710V


5692
675
52
278
31
27
4
100
3
15
15
1
21
5
2
0
0
1
0
3
0
0
0
6
0
2
0
0
Percent of Generated Symbols
0
10
20
30
40
50
60
70
80
90
1
2
3
4
5
6
7
8
9
10
20
30
40
50
60
70
80
90
100
200
300
400
500
600
700
800
900
> 1000
Number of Symbol Replications

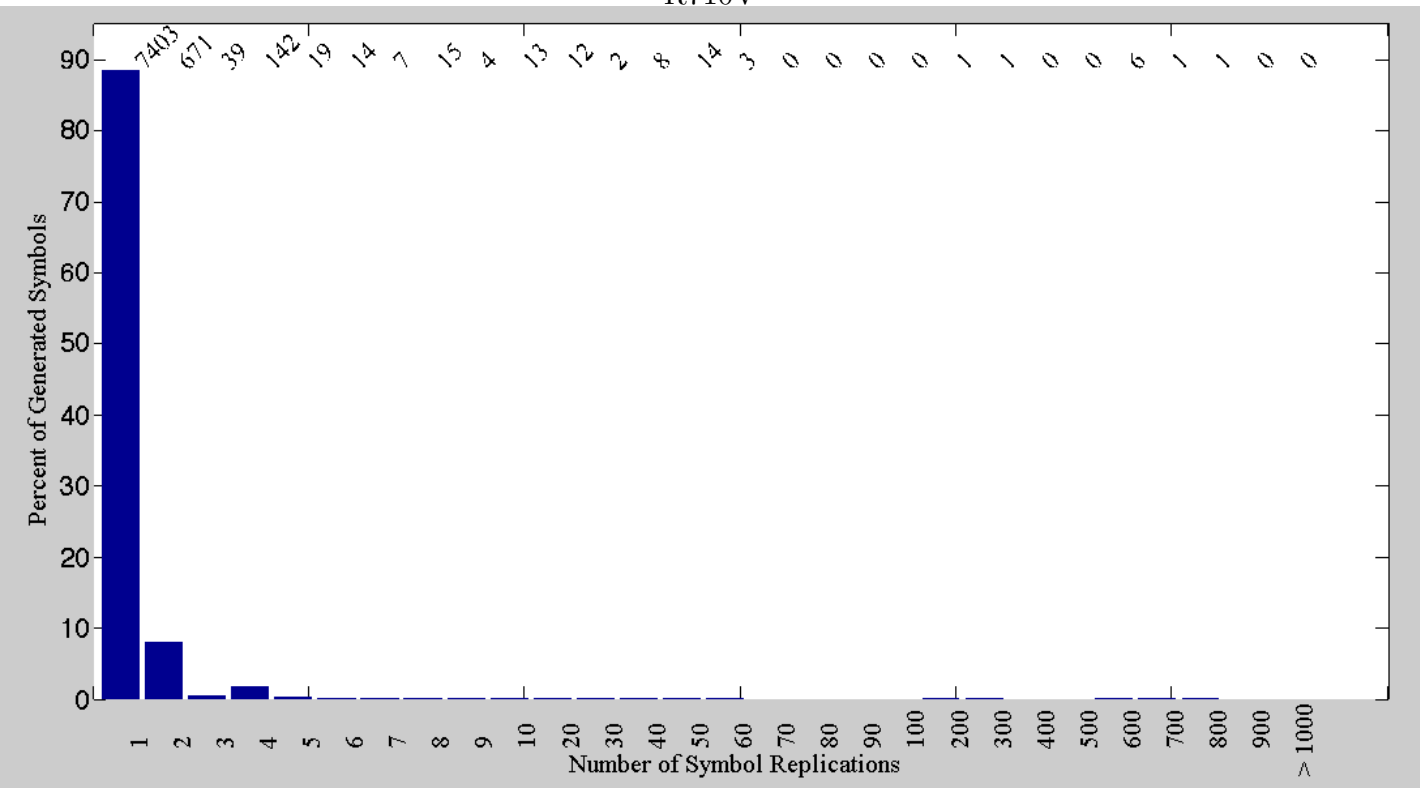
R71V


675
798
94
602
39
92
18
151
10
37
127
19
10
11
10
4
1
1
0
1
0
0
0
7
1
0
0
0
Percent of Generated Symbols
0
5
10
15
20
25
30
1
2
3
4
5
6
7
8
9
10
20
30
40
50
60
70
80
90
100
200
300
400
500
600
700
800
900
> 1000
Number of Symbol Replications

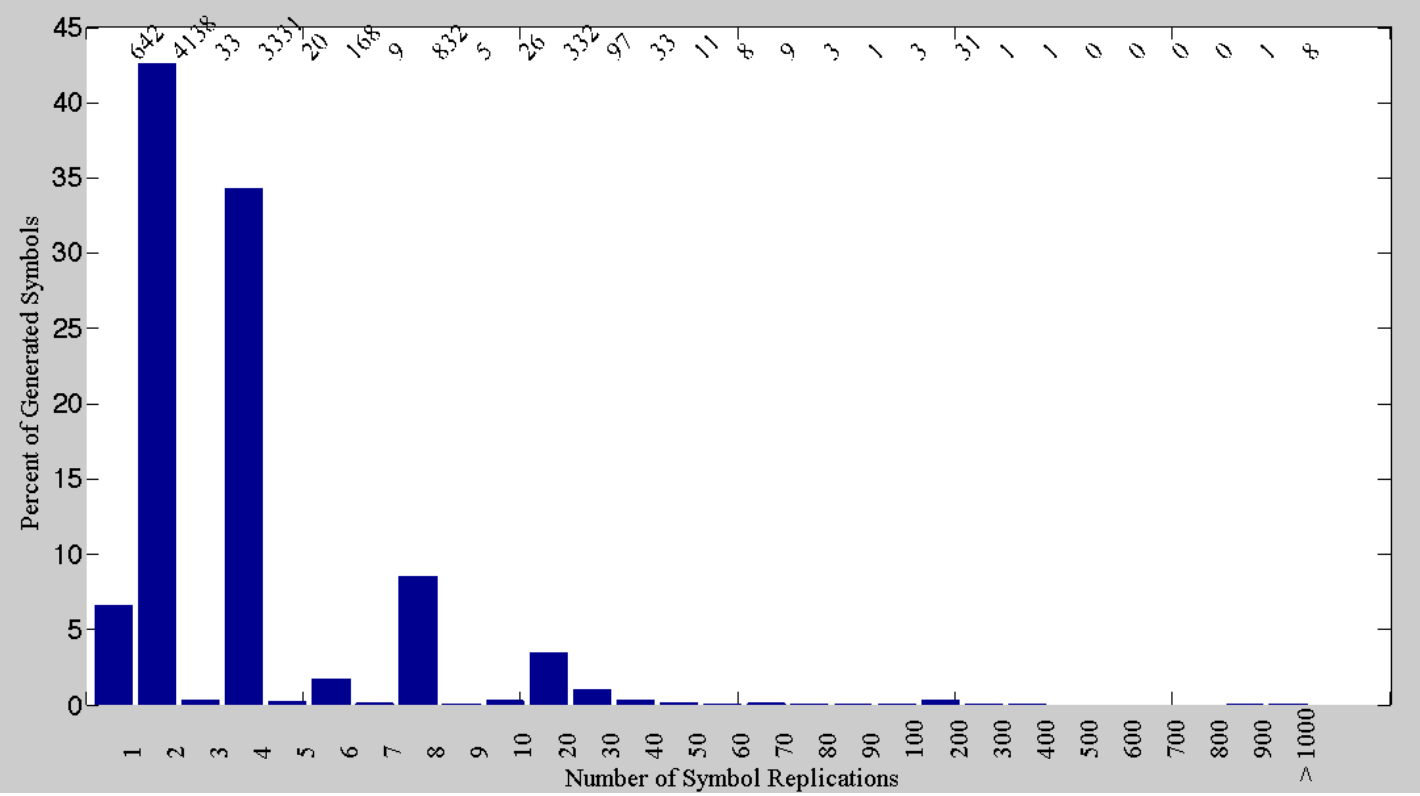
R75V


Percent of Generated Symbols
Number of Symbol Replications
6851
780
114
208
29
23
9
24
10
17
14
1
2
15
8
1
0
0
0
0
0
0
0
8
0
0
0
0
1
2
3
4
5
6
7
8
9
10
20
30
40
50
60
70
80
90
100
200
300
400
500
600
700
800
900
> 1000

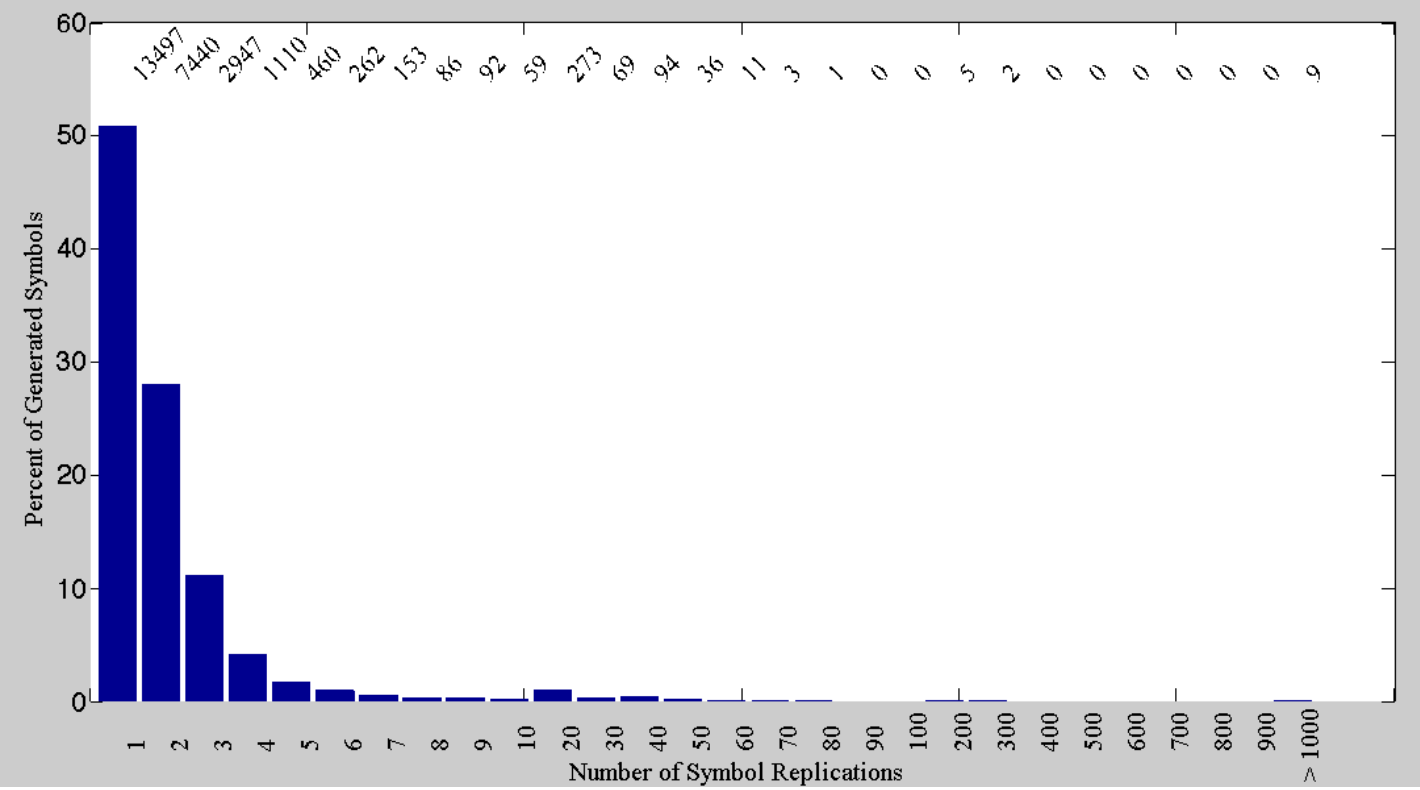
R710V


Percent of Generated Symbols
Number of Symbol Replications
7403
671
39
142
19
14
7
15
4
13
12
2
8
14
3
0
0
0
0
1
1
0
0
6
1
1
0
0
1
2
3
4
5
6
7
8
9
10
20
30
40
50
60
70
80
90
100
200
300
400
500
600
700
800
900
> 1000

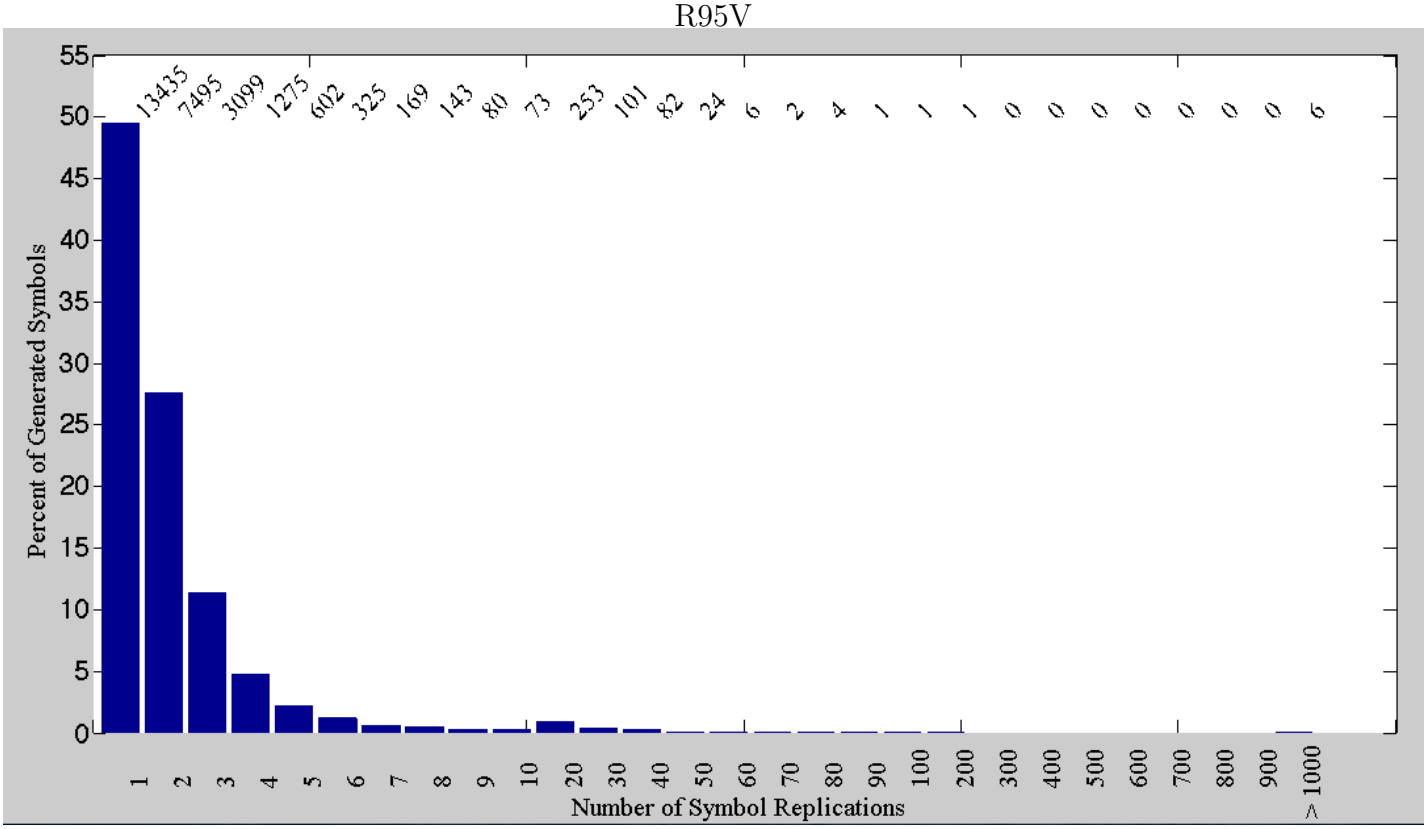
L810V


642
4138
33
3331
20
168
9
832
5
26
332
97
33
11
8
9
3
1
3
31
1
1
0
0
0
0
1
8
Percent of Generated Symbols
0
5
10
15
20
25
30
35
40
45
1
2
3
4
5
6
7
8
9
10
20
30
40
50
60
70
80
90
100
200
300
400
500
600
700
800
900
> 1000
Number of Symbol Replications

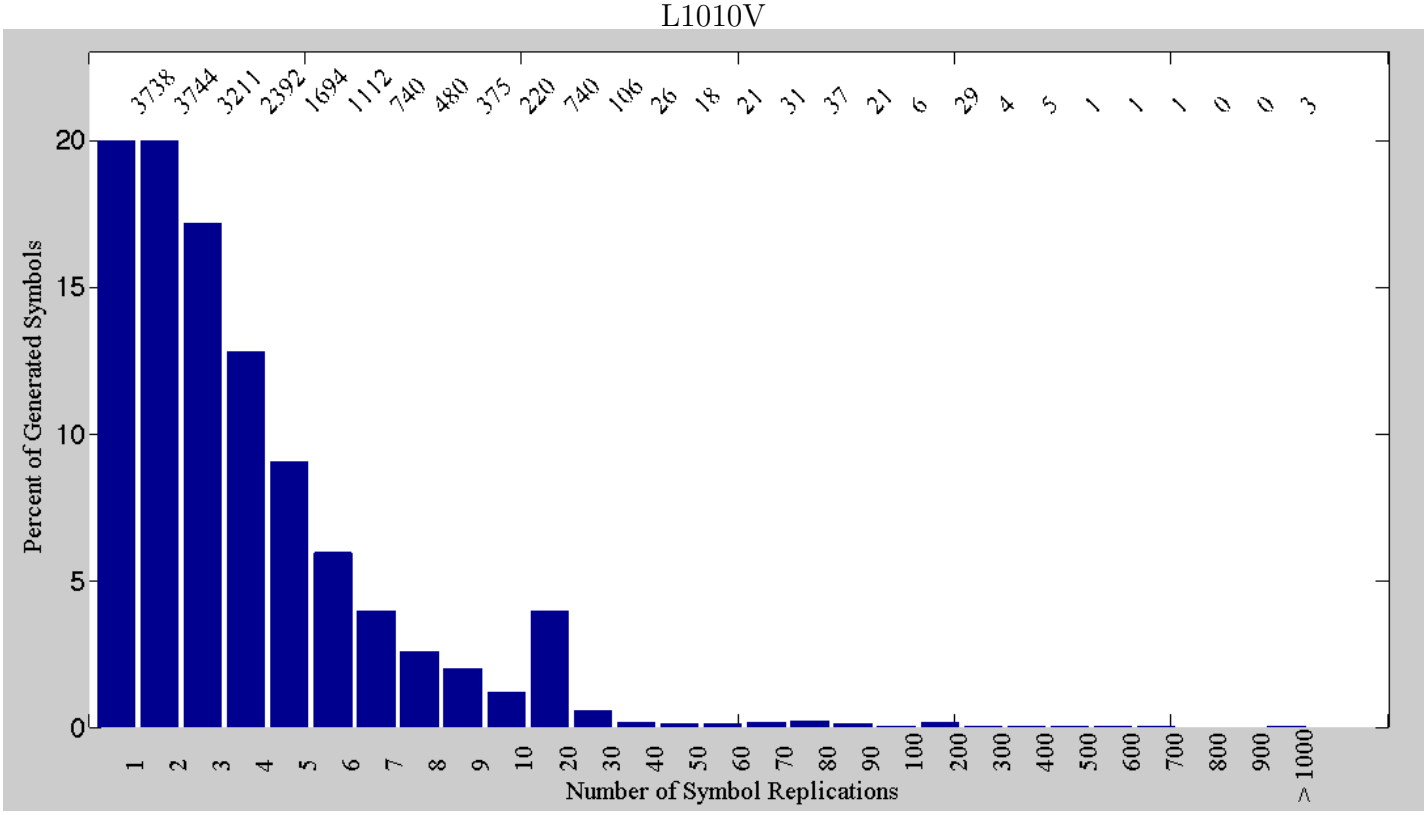
L910V


13497
7440
2947
1110
460
262
153
86
92
59
273
69
94
36
11
3
1
0
0
5
2
0
0
0
0
0
0
9
Percent of Generated Symbols
0
10
20
30
40
50
60
1
2
3
4
5
6
7
8
9
10
20
30
40
50
60
70
80
90
100
200
300
400
500
600
700
800
900
> 1000
Number of Symbol Replications

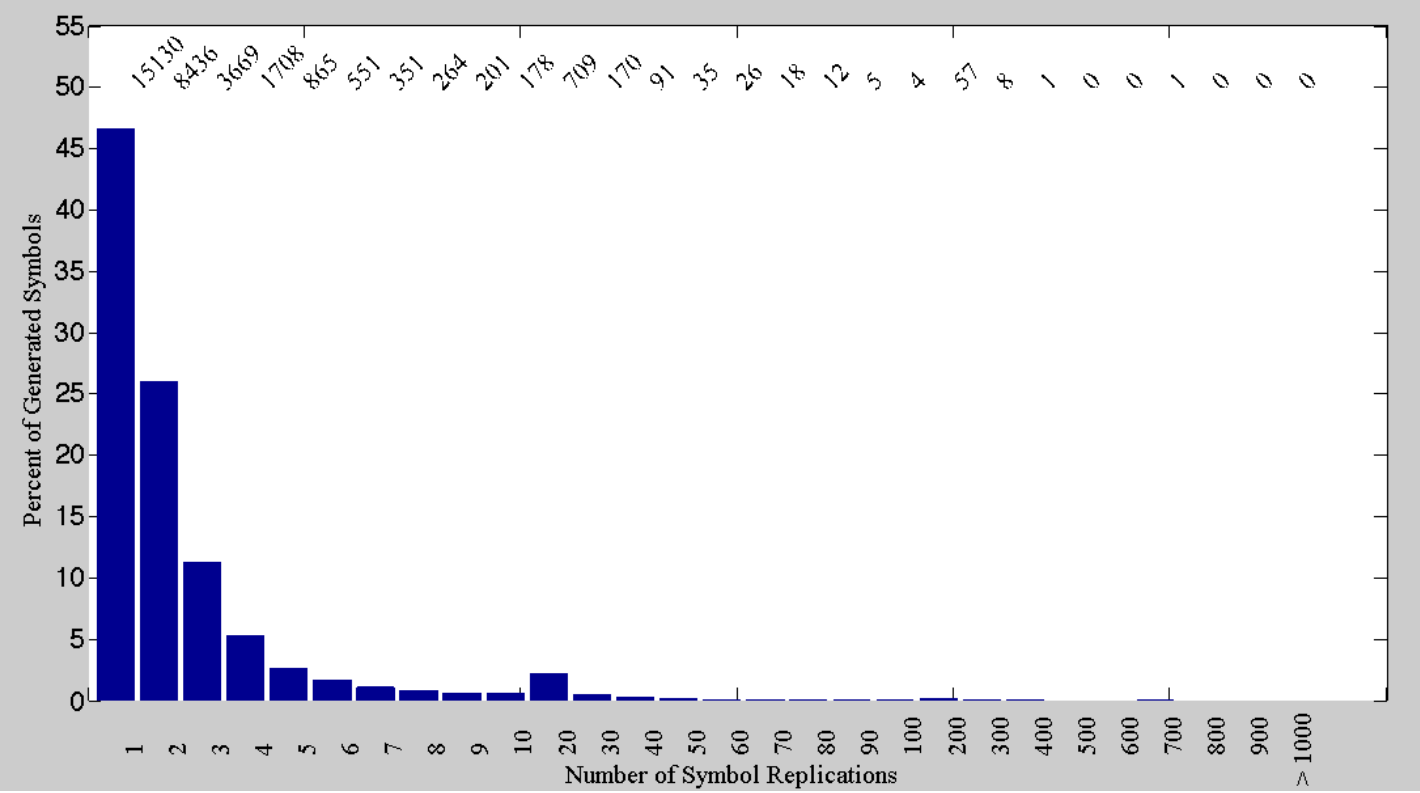
R95V


13435
7495
3099
1275
602
325
169
143
80
73
253
101
82
24
6
2
4
1
1
1
0
0
0
0
0
0
0
6
Percent of Generated Symbols
0
5
10
15
20
25
30
35
40
45
50
55
1
2
3
4
5
6
7
8
9
10
20
30
40
50
60
70
80
90
100
200
300
400
500
600
700
800
900
> 1000
Number of Symbol Replications

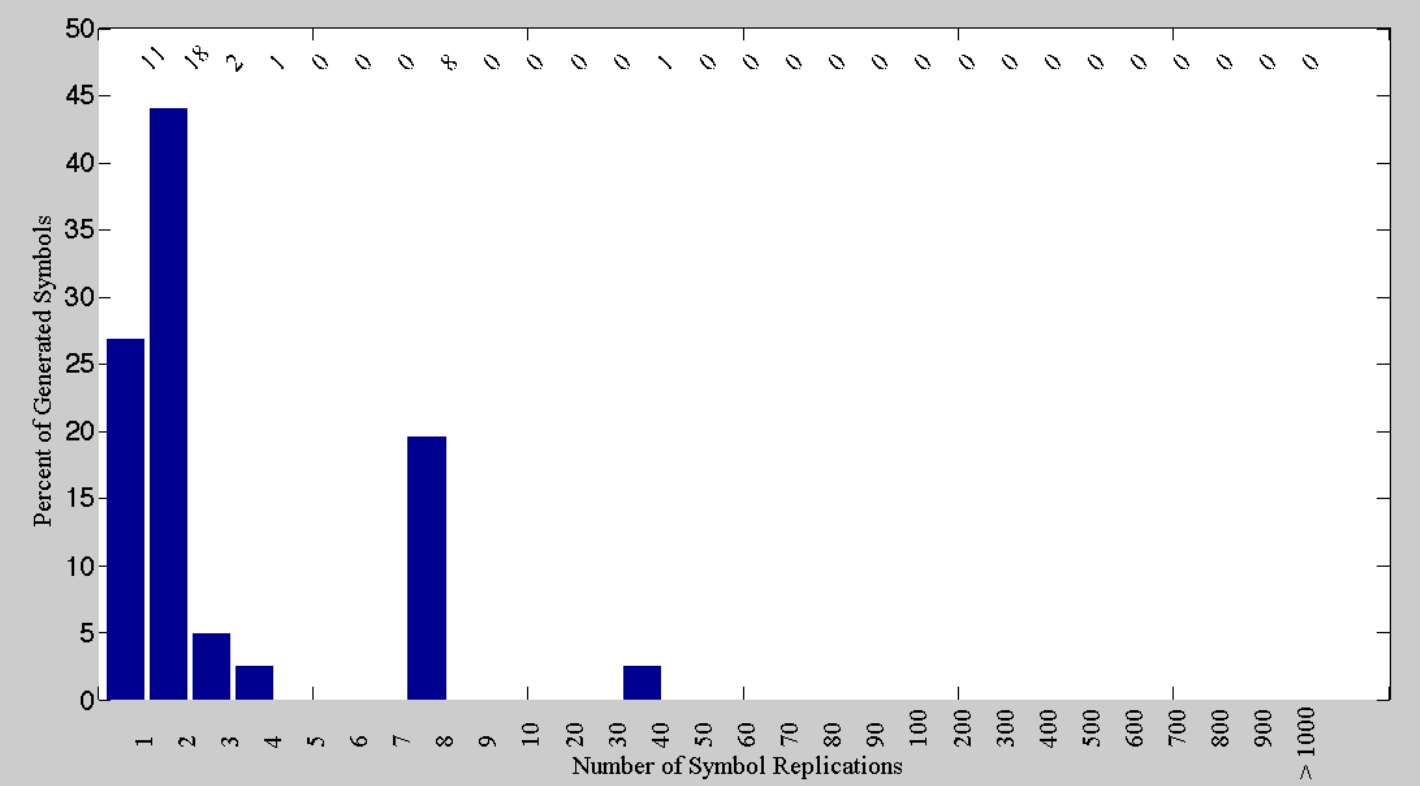
L1010V


3738
3744
3211
2392
1694
1112
740
480
375
220
740
106
26
18
21
31
37
21
6
29
4
5
1
1
1
0
0
3
Percent of Generated Symbols
0
5
10
15
20
1
2
3
4
5
6
7
8
9
10
20
30
40
50
60
70
80
90
100
200
300
400
500
600
700
800
900
> 1000
Number of Symbol Replications

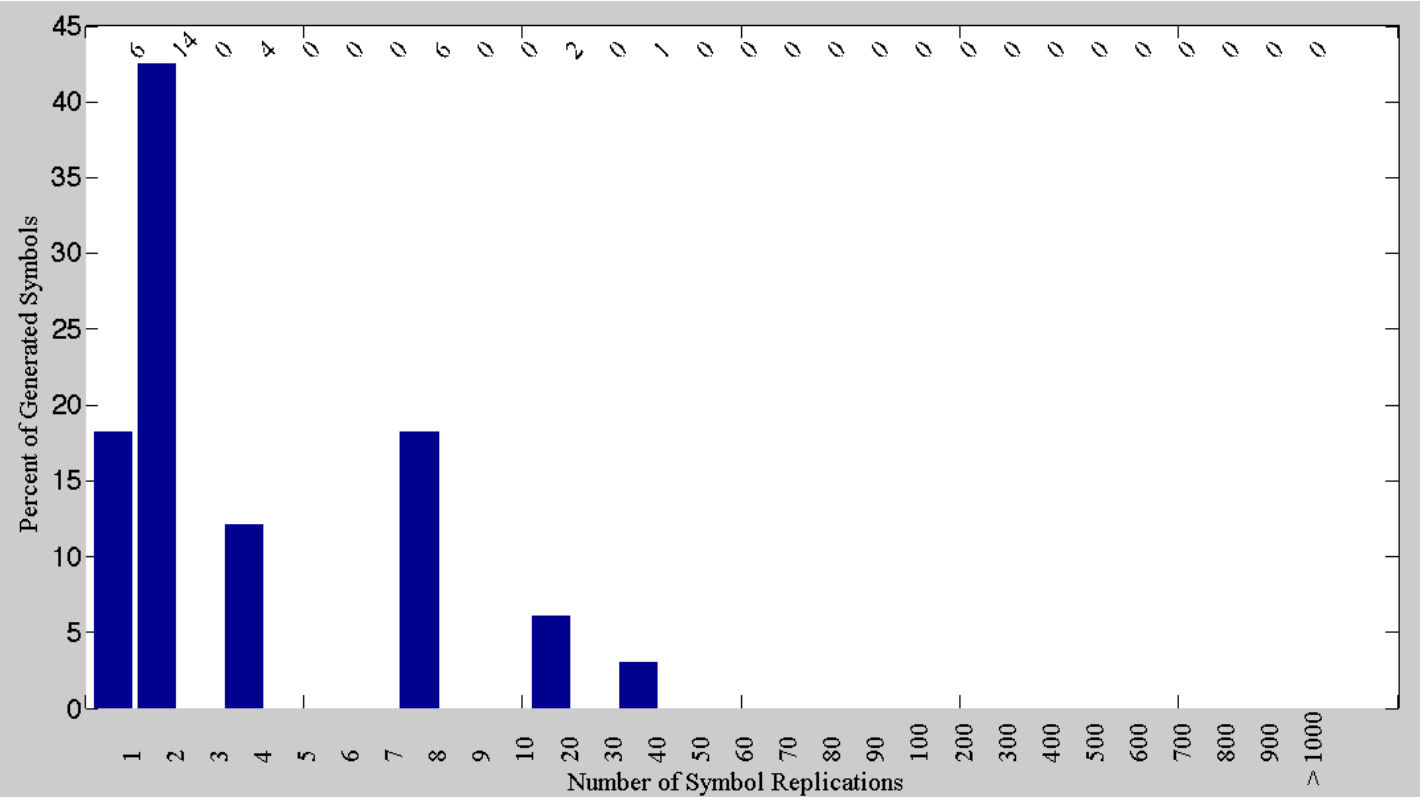
R105V


Percent of Generated Symbols
Number of Symbol Replications
15130
8436
3669
1708
865
551
351
264
201
178
709
170
91
35
26
18
12
5
4
57
8
1
0
0
1
0
0
0
1
2
3
4
5
6
7
8
9
10
20
30
40
50
60
70
80
90
100
200
300
400
500
600
700
800
900
> 1000

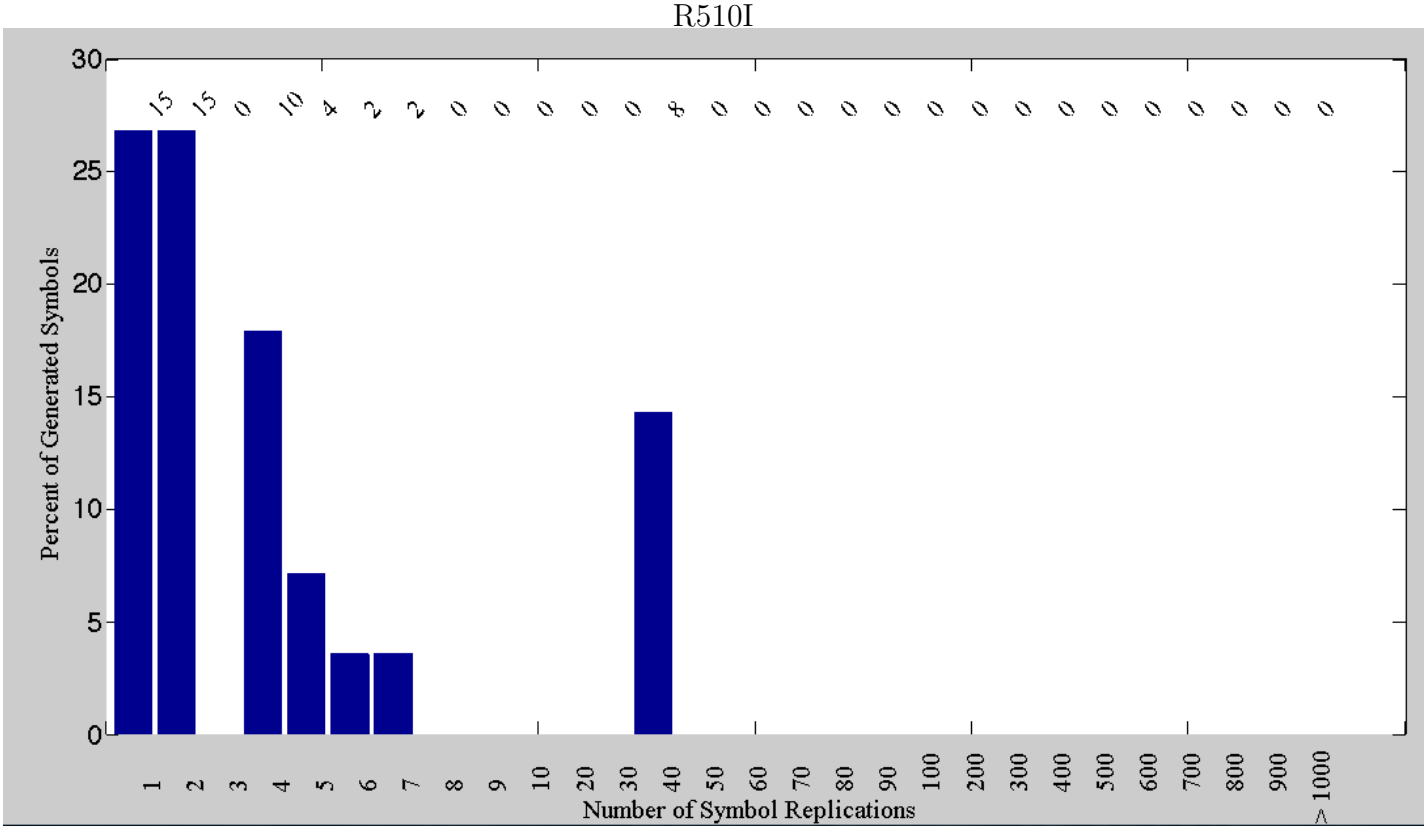
R410I


Percent of Generated Symbols
Number of Symbol Replications
11 18 2 1 0 0 0 8 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 2 3 4 5 6 7 8 9 10 20 30 40 50 60 70 80 90 100 200 300 400 500 600 700 800 900 > 1000

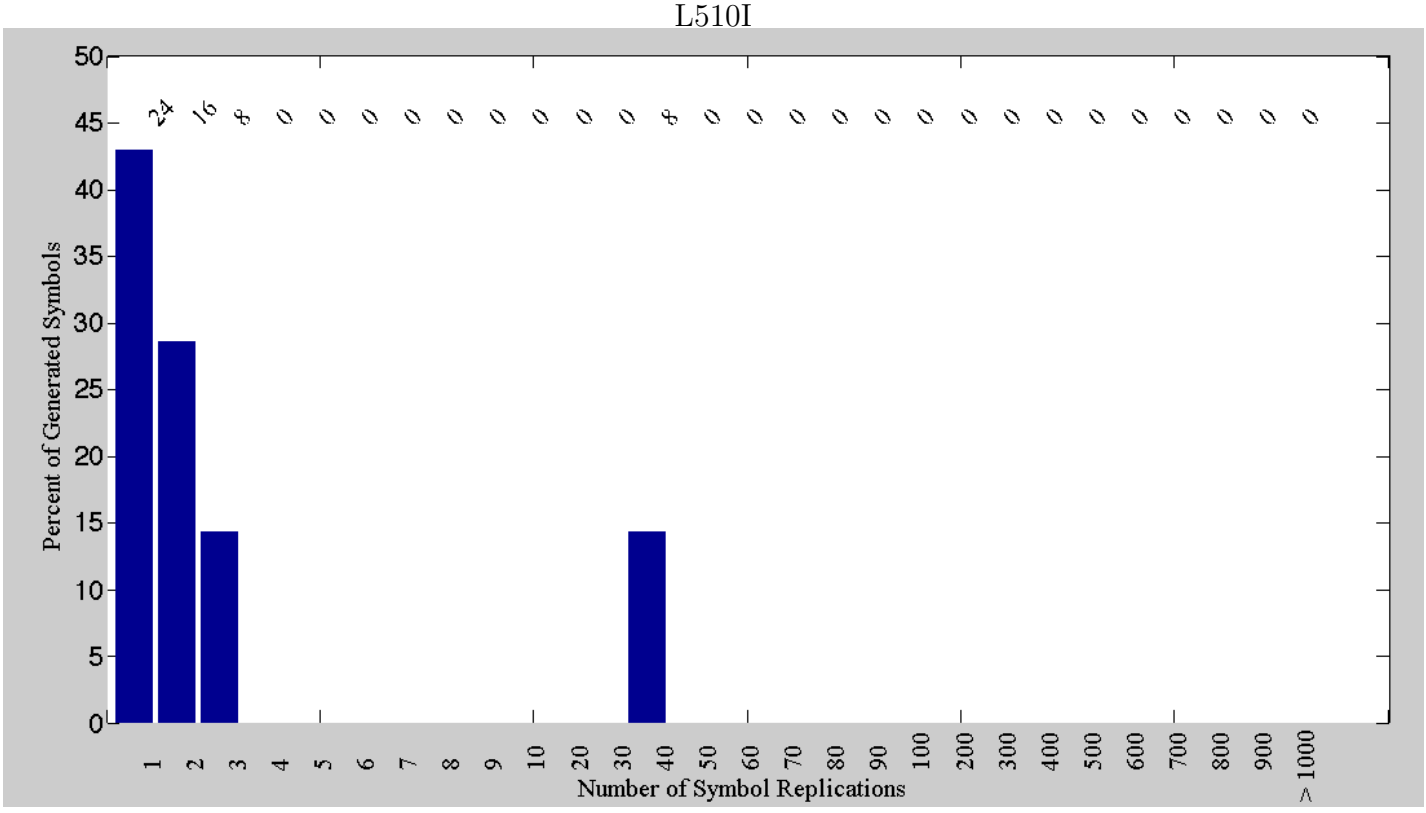
L410I


Percent of Generated Symbols
Number of Symbol Replications
6 14 0 4 0 0 0 6 0 0 2 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 2 3 4 5 6 7 8 9 10 20 30 40 50 60 70 80 90 100 200 300 400 500 600 700 800 900 > 1000

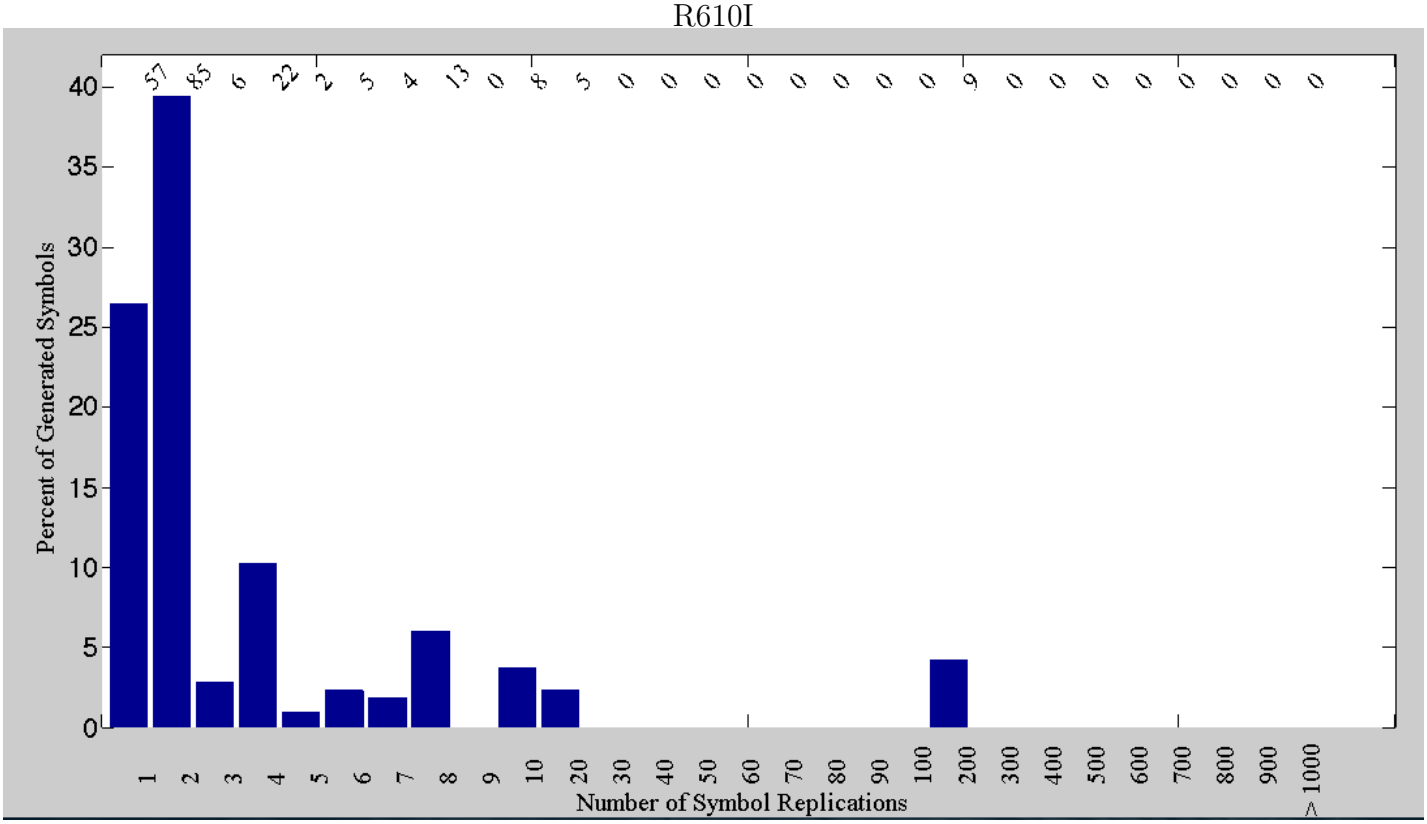
R510I


Percent of Generated Symbols
Number of Symbol Replications
0
5
10
15
20
25
30
1
2
3
4
5
6
7
8
9
10
20
30
40
50
60
70
80
90
100
200
300
400
500
600
700
800
900
> 1000
15 15 0 10 4 2 2 0 0 0 0 0 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0

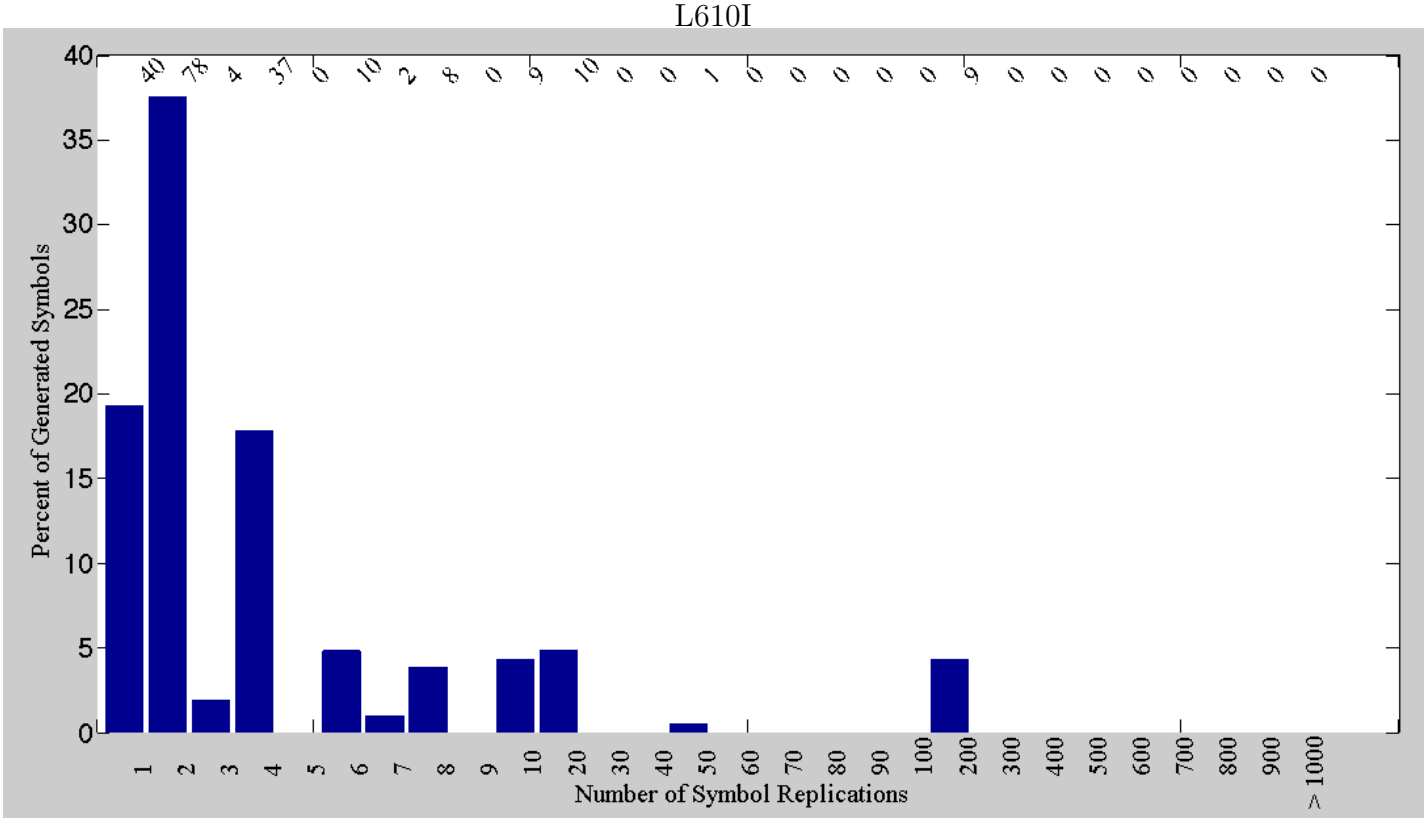
L510I


Percent of Generated Symbols
Number of Symbol Replications
0
5
10
15
20
25
30
35
40
45
50
1
2
3
4
5
6
7
8
9
10
20
30
40
50
60
70
80
90
100
200
300
400
500
600
700
800
900
> 1000
24 16 8 0 0 0 0 0 0 0 0 0 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0

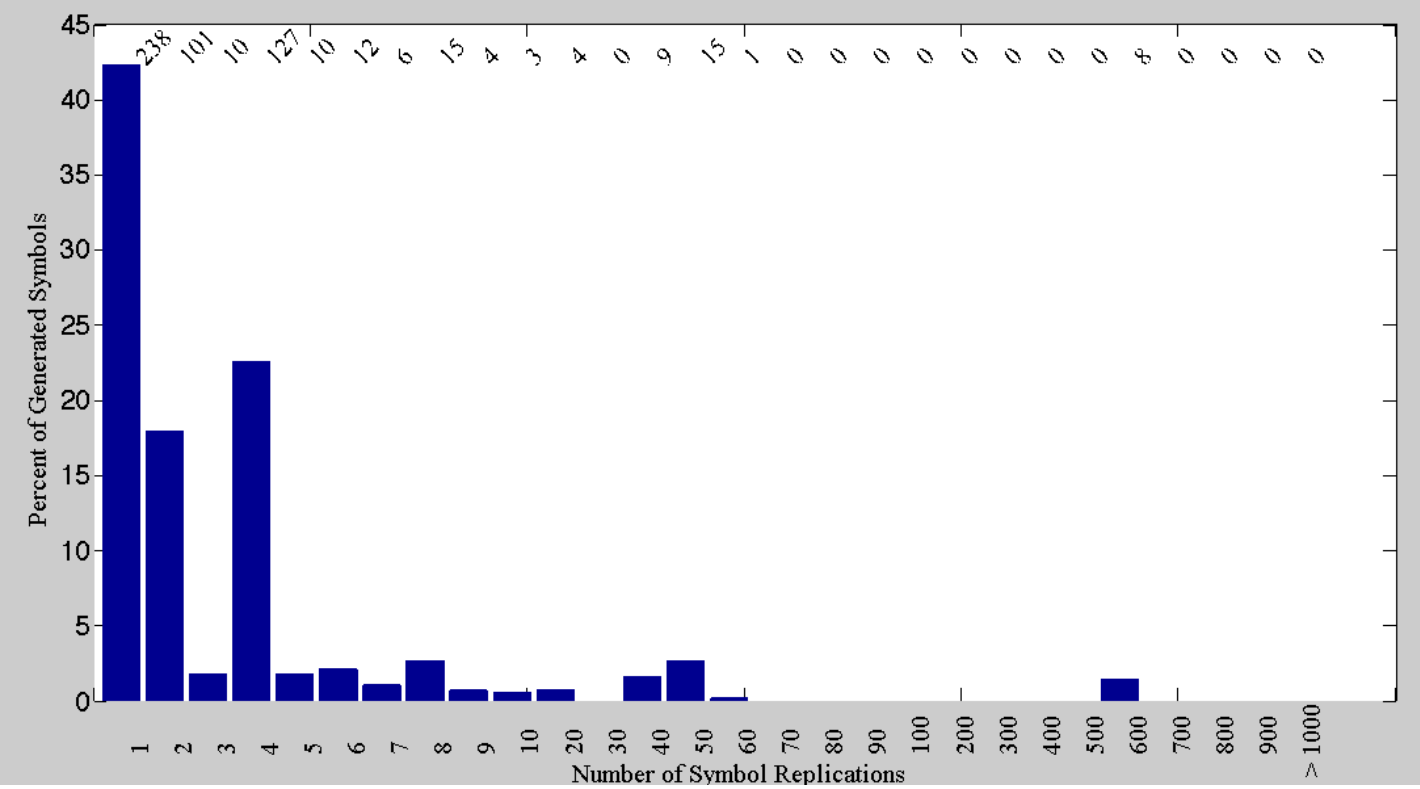
R610I


Percent of Generated Symbols
Number of Symbol Replications
0
5
10
15
20
25
30
35
40
1
2
3
4
5
6
7
8
9
10
20
30
40
50
60
70
80
90
100
200
300
400
500
600
700
800
900
> 1000
57
85
6
22
2
5
4
13
0
8
5
0
0
0
0
0
0
0
0
9
0
0
0
0
0
0
0
0

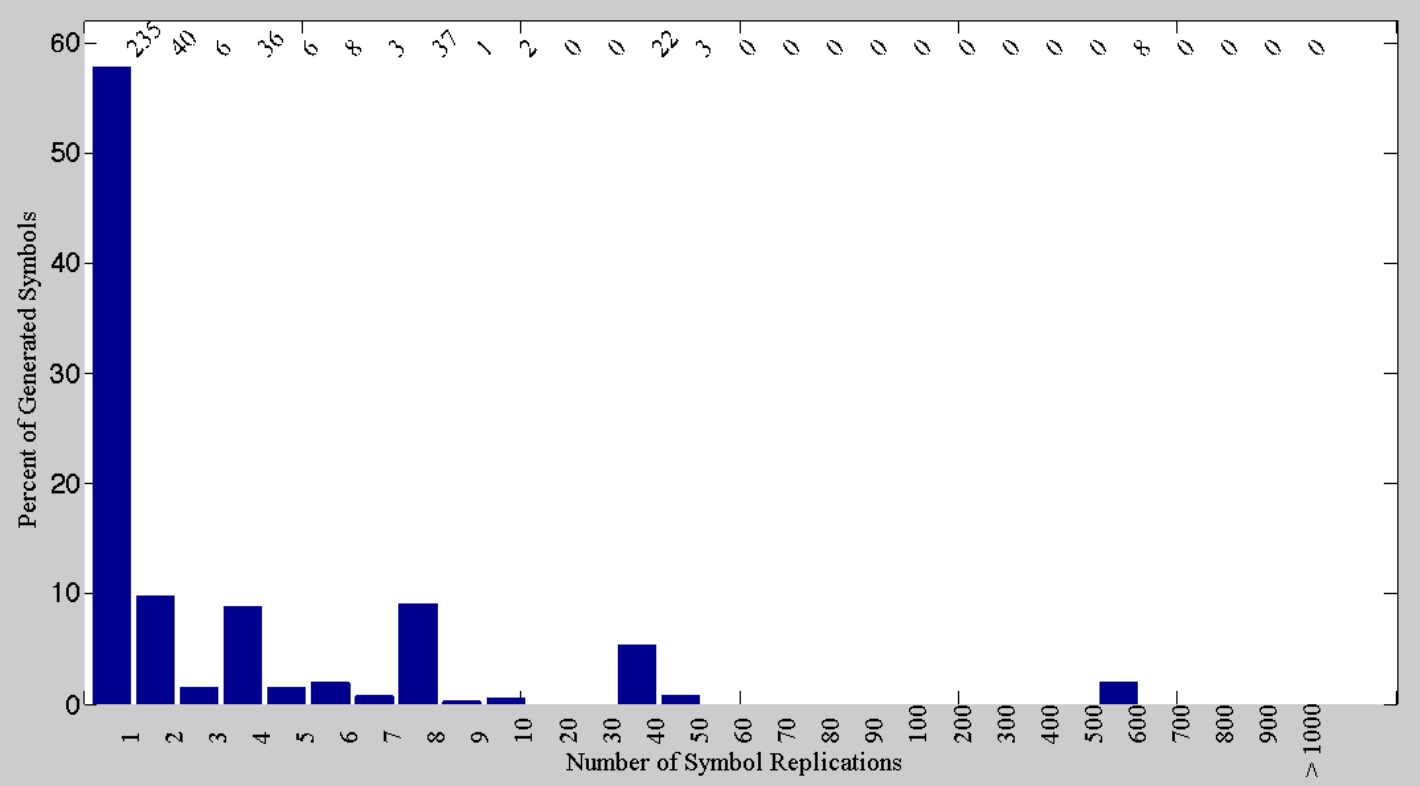
L610I


Percent of Generated Symbols
Number of Symbol Replications
0
5
10
15
20
25
30
35
40
1
2
3
4
5
6
7
8
9
10
20
30
40
50
60
70
80
90
100
200
300
400
500
600
700
800
900
> 1000
40
78
4
37
0
10
2
8
0
9
10
0
0
1
0
0
0
0
0
9
0
0
0
0
0
0
0
0

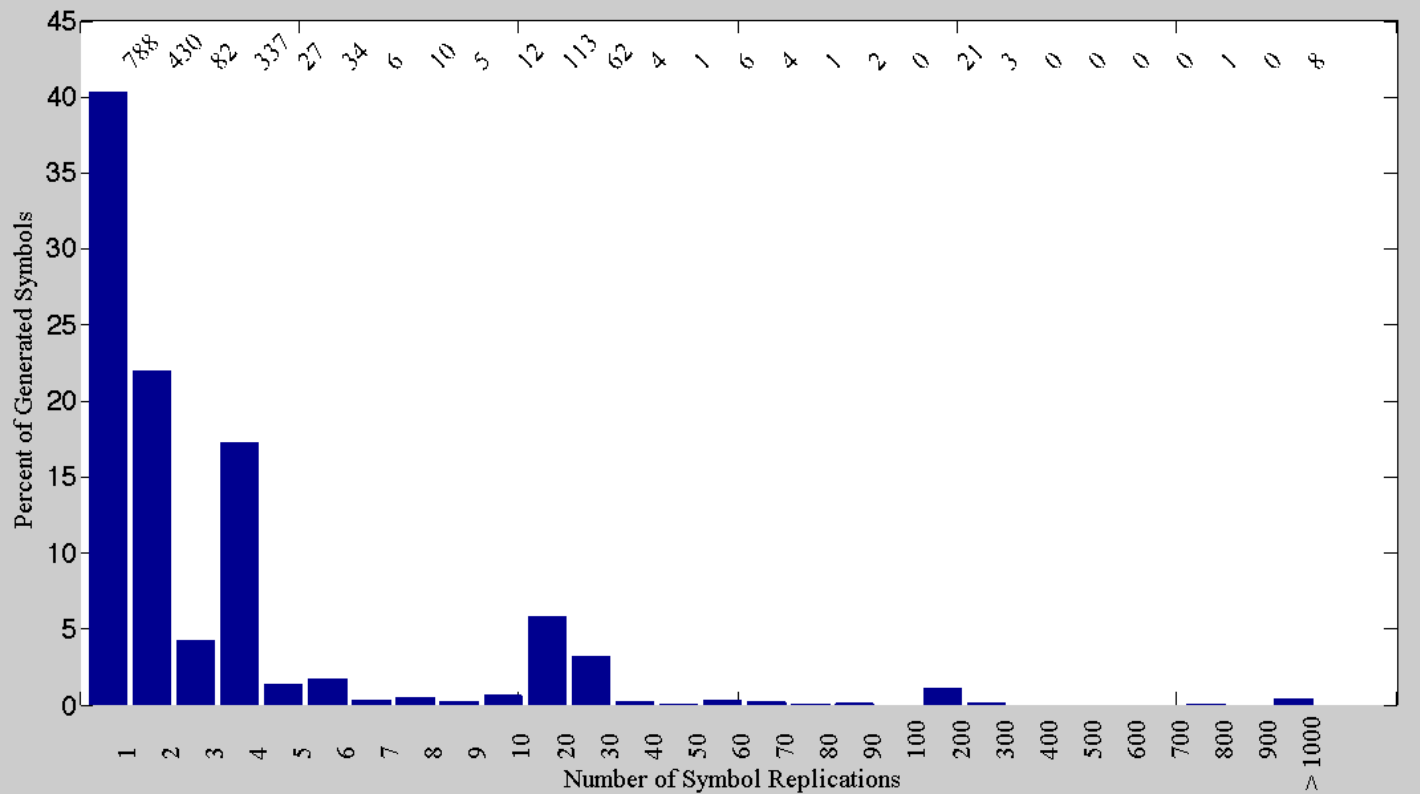
R710I


Percent of Generated Symbols
Number of Symbol Replications
238 101 10 127 10 12 6 15 4 3 4 0 9 15 1 0 0 0 0 0 0 0 0 8 0 0 0 0
1 2 3 4 5 6 7 8 9 10 20 30 40 50 60 70 80 90 100 200 300 400 500 600 700 800 900 > 1000

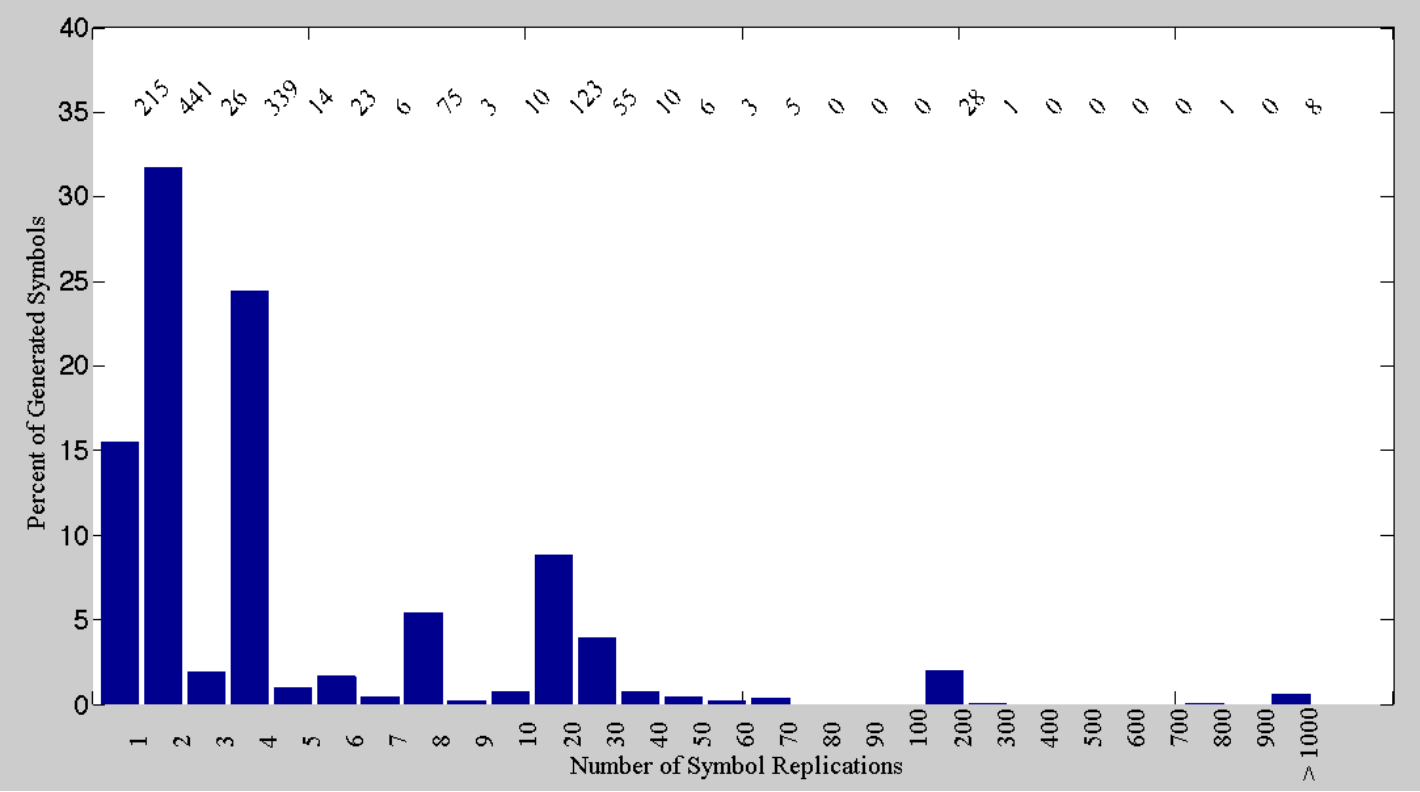
L710I


Percent of Generated Symbols
Number of Symbol Replications
235 40 6 36 6 8 3 37 1 2 0 0 22 3 0 0 0 0 0 0 0 0 0 8 0 0 0 0
1 2 3 4 5 6 7 8 9 10 20 30 40 50 60 70 80 90 100 200 300 400 500 600 700 800 900 > 1000

R810I


788 430 82 337 27 34 6 10 5 12 113 62 4 1 6 4 1 2 0 21 3 0 0 0 0 1 0 8
Percent of Generated Symbols
0 5 10 15 20 25 30 35 40 45
1 2 3 4 5 6 7 8 9 10 20 30 40 50 60 70 80 90 100 200 300 400 500 600 700 800 900 > 1000
Number of Symbol Replications

L810I


215 441 26 339 14 23 6 75 3 10 123 55 10 6 3 5 0 0 0 28 1 0 0 0 0 1 0 8
Percent of Generated Symbols
0 5 10 15 20 25 30 35 40
1 2 3 4 5 6 7 8 9 10 20 30 40 50 60 70 80 90 100 200 300 400 500 600 700 800 900 > 1000
Number of Symbol Replications