Evolvability and Redundancy in Shared Grammar Evolution

Martin H. Luerssen, *Member, IEEE*, and David M. W. Powers, *Senior Member, IEEE*

***Abstract*— Shared grammar evolution (SGE) is a novel scheme for representing and evolving a population of variable-length programs as a shared set of grammatical productions. Productions that fail to contribute to selected solutions can be retained for several generations beyond their last use. The ensuing redundancy and its effects are assessed in this paper on two circuit design tasks associated with random number generation: finding a recurrent circuit with maximum period, and reproducing a De Bruijn counter from a set of seed/output pairs. In both instances, increasing redundancy leads to significantly higher success rates, outperforming comparable increases in population size. The results support previous studies that have shown that representational redundancy can be beneficial to evolutionary search. However, redundancy promotes an increase in further redundancy by encouraging the creation of large offspring, the evaluation of which is computationally costly. This observation should generalize to any unconstrained variable-length representation and therefore represents a notable drawback of redundancy in evolution.**

I. Introduction

Nature's remarkable ability to adapt to cataclysmic changes can be strongly attributed to its diversity. Diversity is equally crucial to the success of artificial evolution. The convergence towards an optimum inherently involves a loss of diversity that can reduce search effectiveness and even stall the process. A large population of solution candidates and a high mutation rate are traditional remedies for this, but since diversity is often contrary to fitness, it rarely survives under selection pressure. Actively selecting towards a diversity objective is one way of addressing this [1], [2], yet perhaps the simplest answer is to just 'hide' the diversity. Having a redundant representation can provide the necessary space for this, which is a strategy that has already shown marked potential in other studies (see Section III for more on this).

Shared Grammar Evolution (SGE) is a new technique for evolving a globally shared repository of grammatical productions from which solution candidates can be derived [3]. Diversity in the repository is essential here, because new solutions can only be created if the necessary productions already exist. The following study explores a simple scheme of keeping productions available for several generations beyond the elimination of the solutions they contributed to. The resulting accumulation of redundancy may constitute a valuable source of diversity, but may also drown out fit building blocks. The viability of the scheme will therefore be evaluated on two circuit design problems related to pseudorandom number generation: determining a circuit that produces a maximum period output, and reverse-engineering a specific circuit, the De Bruijn counter.

Section II introduces the SGE scheme, followed in section III by a discussion of how redundancy may facilitate evolvability in SGE . Sections IV and V address the chosen problem tasks and experimental setup in detail, while the final two sections analyze and summarize the results of the experiment.

This work was in part supported by the 2006/2007 Flinders University Faculty of Science and Engineering Program Grant.

The authors are with the Artificial Intelligence Laboratory, School of Informatics and Engineering, Flinders University of South Australia, Bedford Park SA 5042, Australia (email: martin.luerssen@flinders.edu.au; powers@ieee.org).

II. Shared Grammar Evolution

Combining a grammar with evolution is nothing new; it has been explored previously in two separate lines of research. Firstly, a grammar may be suited as a model of a development system, as reflected in the biological mapping from genotype to phenotype. Lindenmayer systems [4] are popular for this and have been used to optimize neural networks and other designs [5], [6]. Evolution applies here to a population of grammars; a single solution can be derived from each. Alternatively and more commonly, grammars are used as a means of syntactic constraint, with the most studied example being Grammatical Evolution (GE) [7], which evolves programs in a language generated by a user-defined grammar. Other systems have taken this idea further by automatically modifying the grammar to improve the search as it progresses, either by learning [8] or evolving [9] the grammar. Evolution applies here to a population of solutions; new solutions are added by stochastically deriving these from a grammar.

SGE constitutes a hybrid of these approaches. In SGE, a user-defined 'template grammar' specifies the available terminals and functions and any syntactic restrictions for these. As with GE, initial solution candidates are derivations from this template grammar. However, each derived solution is then represented by another 'individual grammar' that is specific to that solution and has no duplicate predecessors – i.e., it is purely deterministic, unlike the template grammar. Throughout subsequent generations, further solutions are obtained by evaluating the effect of random changes to the individual grammars of existing solutions. For our purposes here, SGE will be used to construct programs and circuits as tree data structures, as with genetic programming (GP) [10]. The successors (right-hand sides) of grammatical productions in SGE therefore consist of a function terminal and one or two nonterminal or variable terminal arguments to this function (see Table I for details).

The SGE method is illustrated in Figure 1. A production is first chosen from an existing solution's individual grammar, which consists of the productions that contribute to this solution. A copy of this production is made and then modified by replacing one of its terminals or nonterminals by an

1-4244-1340-0/07/$25.00 ©2007 IEEE

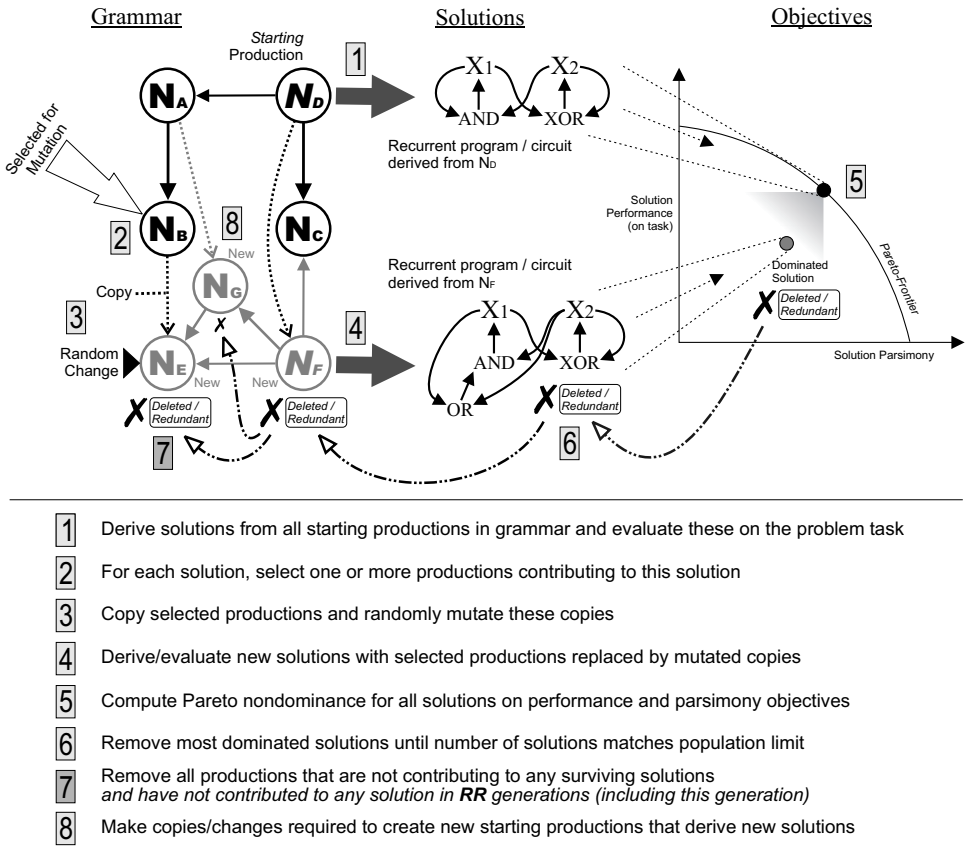

Fig. 1. Each generation of SGE consists of up to 8 main steps and involves the addition and removal of productions from a shared grammar set based on the performance and size of the solutions to which they contribute. Production redundancy arises when non-contributing productions are retained (at step 7).

alternative – in the case of a terminal, this is from the original template grammar; in the case of a nonterminal, it can be a reference to any production of any other solution's grammar or the template grammar. A new solution is then derived from the original solution's grammar with the modified production expressed in place of the original production. Infinite recursion is prevented by associating each production with a recursion limit value, which defines the maximum recursion depth of a production calling itself; if the limit is exceeded during derivation, the production is replaced by an associated default terminal (see Table I).

If after evaluation on the objective function the new solution is sufficiently fit to be selected into the next generation, then a new individual grammar is created for this solution. Any production that needs to refer to any modified productions is copied and modified accordingly. Unchanged productions are not copied, but referenced directly, so it is possible that any one production may contribute to several other solutions. We therefore regard the global set of all productions defining all solutions, i.e., all individual grammars, as a shared grammar set. Depending on production reuse, this shared grammar set may be considerably smaller than the collective size of all solutions.

Previously explored benefits of SGE are performance benefits from sharing subsolutions results between solutions [3] and extending the grammar to more complex data structures such as graphs [11]. A drawback of SGE is that cross-referencing between productions can be extensive, leading to rapid growth in solution size beyond what is necessary for problem solving. This problem is to a lesser extent also encountered in GP and known as bloat [12]. SGE addresses it by employing a multi-objective evolutionary algorithm (MOEA), based on the NSGA-II presented by Deb *et al.* [13], to explicitly determine the trade-off between program performance and size, an approach that has also been successful for GP [1].

III. Evolvability

Evolvability is the capacity of an evolutionary system to continuously produce and maintain potentially adaptive variants of solutions [14]. It necessitates that a balance is found between exploitation, i.e., greedy search, and exploration,

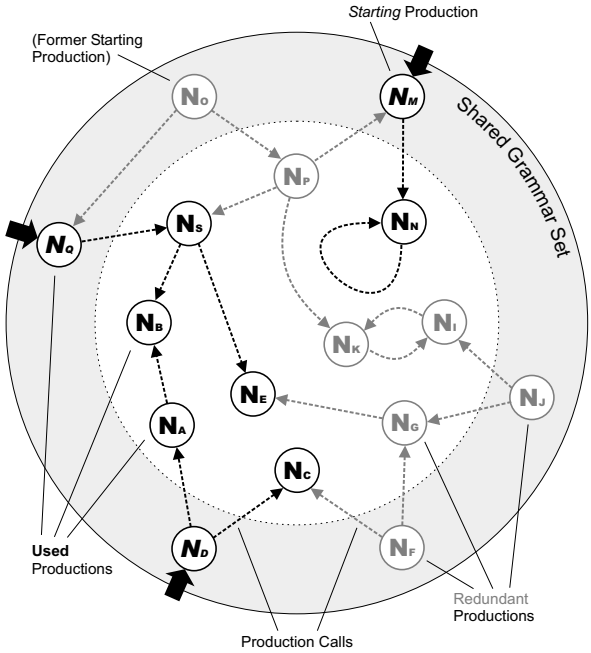

Fig. 2. Solutions constitute specific production sequences within the shared grammar set. Without redundancy, new solutions must be defined either by productions that are part of an existing solution or by mutations thereof. With redundancy, the choice is extended to productions that do not presently contribute to a solution, but have done so previously. This allows for large production sequences, i.e., building blocks, to accumulate and be integrated into new solutions. A redundant production is removed when it has not contributed to a solution for a number of generations specified by the *RR* parameter.

i.e., diversity in the population. Changes to population size, mutation rates, and selection schemes have been the standard way of addressing this. More recently, evolvability has been linked to redundancy in the representation space – also known as fitness neutrality, since changes to the redundant representation have no impact on selection. Redundancy in evolutionary computation has been the subject of several studies, some of which determined that redundancy leads to improve evolvability [15]–[21], while others did not [22], [23]. The discrepancy has been blamed on a poor understanding of what redundancy means in this context [24]. Redundancy has been suggested to be beneficial only if it does not actually code for any phenotypic trait, but instead lies dormant in the representation [21]. Hidden variation may thus accumulate as redundant code and, once revealed, fuel the kind of rapid adaptation needed to escape from any suboptima that have trapped the search [14].

In biological genetics, redundancy occurs either by negative linkage disequilibrium, where alleles of opposite effect occur together and cancel each other out, or by canalization, where the effects of alleles are reduced by down-regulation of their expression. In the SGE framework, the former would depend on whether the problem task makes it likely for subprograms to cancel each other out, and therefore would be difficult to vary independently. On the other hand, canalization does not occur at all, because those productions that are called are expressed and those that are not are removed – but we can change this.

Simply not removing productions would lead to an accumulation of potential building blocks that exist beyond the selection constraints. Since such a grammar growth would not be sustainable over many generations, we suggest that productions are ultimately deleted, but with a delay of a user-defined number of generations subsequent to their last expression. This includes productions created for unfit solution candidates, so even a short delay leads to a substantial build-up.

In SGE, minor changes to a production can lead to substantial changes in the derived solution, as an entirely different sequence of production calls might follow. The called productions need to exist, however, and it is here that redundant productions improve the diversity of possible choices, potentially leading to more diverse and fitter solutions. It is likewise conceivable that the redundancy, which is not subject to selection, contains little that is useful for a fit solution. Our experiment below is an attempt to clarify this issue.

IV. Pseudorandom Number Generation

Random numbers are required for a wide range of important applications such as data encryption and also play an essential part in evolutionary algorithms. Random numbers are typically obtained from pseudorandom number generators (PRNGs) implemented either in software or directly in hardware. Since complex arithmetic operations are often not feasible in hardware, it is desirable for PRNGs to be based on hardware friendly operations, i.e., strictly Boolean operators.

As the problem task for evaluating the impact of redundancy on SGE, we chose a circuit design problem closely related to PRNGs. The pattern generated by a PRNG repeats itself after a certain number of cycles, known as the period of the generator. PRNGs with short periods are easy to predict, so our objective is to design a circuit that produces the maximum possible period – which is equal to the number of distinct states of the circuit. For recurrent circuits with 4 or 8 binary variables, the maximum period would be $2^4 = 16$ and $2^8 = 256$, respectively.

A high period circuit is not the same as a PRNG, since the state may still be easy to predict, e.g., in the case of an incrementing circuit. Design of a PRNG would typically involve evaluation of possible solutions against a rigorous statistical test, such as the DIEHARD suite [25] (which we intend to address in a later paper). Design of a high period circuit still allows for a wide range of complex solutions while also being inexpensive to evaluate.

Conversely, it is also possible to determine a specific PRNG from a random sequence. This is a more challenging problem, as it is unlikely to allow for more than one solution. We chose a standard PRNG, the 4-bit De Bruijn counter shown in Figure 3 (center), as the target for this. A perfect

TABLE I
DEFAULT PARAMETERS FOR THE TWO MAIN PROBLEM TASKS.

	4(8)-Register Maximum Period Circuit	4-bit De Bruijn Counter
Objective	Design a recurrent Boolean circuit with 4 (8) registers that produces the maximum possible period	Design a recurrent Boolean circuit that functionally matches a De Bruijn Counter
Terminals	AND, OR, XOR, NOT and a register for each binary variable (available as inputs and outputs of the program defining the circuit); the default terminal for all functions is 0/FALSE	
Fitness Case(s)	No register state repeated for 16 cycles (256 for 8-register circuit), starting from zero seed	All 16 possible seeds (and resulting bit sequences) for the De Bruijn Counter
Simulation	Simulation for 16 cycles (256 for 8-register circuit)	Simulation for 20 cycles (16 + 4 to verify return to initial seed)
Error Measure	Number of cycles before state is repeated	Proportion of incorrectly reproduced bits
Size Measure	Number of expressed productions/terminals; solutions are invalid if consisting of more than 1000 terminals	
Mutation	A single production is selected for mutation and a single nonterminal or terminal of this production is replaced by an alternative, at 33% chance from the template grammar, at 66% chance from the global grammar. Additionally, there is a one in six chance that the recursion limit of a production is increased or decreased by one.	
Termination	After 1000 generations.	
Experiments	50 independent runs performed for each parameter setting (see text).	

solution should match the output of the De Bruijn counter for each possible initial state, or *seed*.

V. EXPERIMENTS

Redundancy in SGE is constituted by unused productions within the shared grammar set. By default, productions remain in the grammar for only the generation in which they contribute to a solution candidate. If we keep the productions until the end of the next generation, then we have a redundant production retention (RR) of two generations. We will increase this retention in powers of two up to 32 generations (marked as RR = 32), leading to large parts of the shared grammar set being redundant.

Any observed changes in performance need not be due to redundancy, however, but could instead be due to the larger total number of productions. For comparison, we hence also evaluate a doubling of the base population size of 5 solutions to up to 160 solutions. A 5 solution shared grammar set with a tolerance of 32 generations defines the same number of solution candidates as a 160 solution shared grammar set. While 155 of these were deselected previously and are not competing in this generation, their contributing (and now mostly redundant) productions are still available for constructing new solution candidates. Unlike an increase in the population size, maintaining redundant productions should not substantially affect performance. However, we noted an acute increase in very large solution candidates with increased redundancy (see below), so parameter combinations were capped to a total number of solutions, actual or implicit, of 160 per generation.

SGE is applied to the problems tasks described in the previous section, which include finding a high period recurrent circuit with 4 or 8 binary registers and a recurrent circuit that reproduces a 4-bit De Bruijn counter for all 16 possible seeds. Available Boolean operators and default parameters for evolution are listed in Table I. Since these problems are concerned with PRNG design, it is worth noting that SGE employs a Mersenne Twister [26] as its internal PRNG.

VI. RESULTS

Finding a recurrent circuit with maximum period output is made difficult by the existence of a simple, suboptimal solution: the Linear Feedback Shift Register (LFSR). The LFSR shifts each of its N bits into adjacent registers, with the first bit defined by the XOR of several of these bits, the so-called 'taps', as illustrated in Figure 3 (left). The largest state space possible for an LFSR is $2^N - 1$, which is one cycle short of the maximum period. LFSRs are commonly employed as basic hardware PRNGs, and in this sense SGE is very effective at evolving a PRNG. 1047 out of $21 \times 50 = 1050$ runs found a 4-register solution with a maximum period of 15 or 16. However, the 16-period circuit is more complex than the LFSR, and with no intermediate solutions between them, overcoming this complexity gap appears to be difficult. The scalability of the maximum period problem is very poor in this respect, as none of runs on the 8-register problem generated any maximum period circuits, although there were $931/1050$ LFSR designs.

The evolution of the 4-register circuit is considerably more successful and informative, as seen in Table II (top). Several maximum period circuits are found, the simplest example of which is shown in Figure 3 (right). Given the choice of permitted gates, this circuit is in fact simpler than the De Bruijn counter (which is also a 2^N period circuit). Conversely, the results reported in Table II (bottom) suggest that it is disproportionally harder to evolve a De Bruijn counter directly, but this is also a more constrained task, as each seed state is expected to produce a specific sequence of bits. Even a valid De Bruijn counter may fail this task if it taps the registers in the wrong order. SGE nevertheless manages to succeed in several instances of correctly determining the De Bruijn counter from the presented sequences, thereby

TABLE II

TOP ROW OF EACH CELL RECORDS THE SUCCESS RATE OF THE EXPERIMENT, BOTTOM ROW SHOWS THE MEAN AND STANDARD DEVIATION OF THE ERROR OF THE BEST SOLUTION. 8-REGISTER RESULTS ARE NOT SHOWN DUE TO CONSISTENT PREMATURE CONVERGENCE (SEE TEXT).

4-Register Maximum Period Circuit

Redundancy Retention \ Population	5	10	20	40	80	160
1	2% 1.04±0.45	4% 0.98±0.25	12% 0.88±0.33	16% 0.84±0.37	40% 0.60±0.49	52% 0.48 ± 0.50
2	8% 0.92±0.27	26% 0.74±0.44	72% 0.28±0.45	88% 0.12±0.33	96% 0.04±0.20	
4	38% 0.62±0.49	64% 0.36±0.48	92% 0.08±0.27	96% 0.04±0.20		
8	54% 0.46±0.50	88% 0.12±0.33	96% 0.04±0.20			
16	72% 0.30±0.51	92% 0.08±0.27				
32	66% 0.34±0.48					

4-Bit De Bruijn Counter

Redundancy Retention \ Population	5	10	20	40	80	160
1	2% 0.27±0.05	2% 0.27±0.05	2% 0.28±0.06	0% 0.28±0.03	4% 0.26±0.06	4% 0.26 ± 0.05
2	4% 0.26±0.06	2% 0.27±0.05	8% 0.24±0.08	28% 0.18±0.11	36% 0.16±0.12	
4	2% 0.27±0.05	4% 0.25±0.07	22% 0.20±0.11	28% 0.18±0.11		
8	6% 0.25±0.08	8% 0.23±0.07	32% 0.17±0.12			
16	4% 0.27±0.07	6% 0.21±0.10				
32	12% 0.23±0.09					

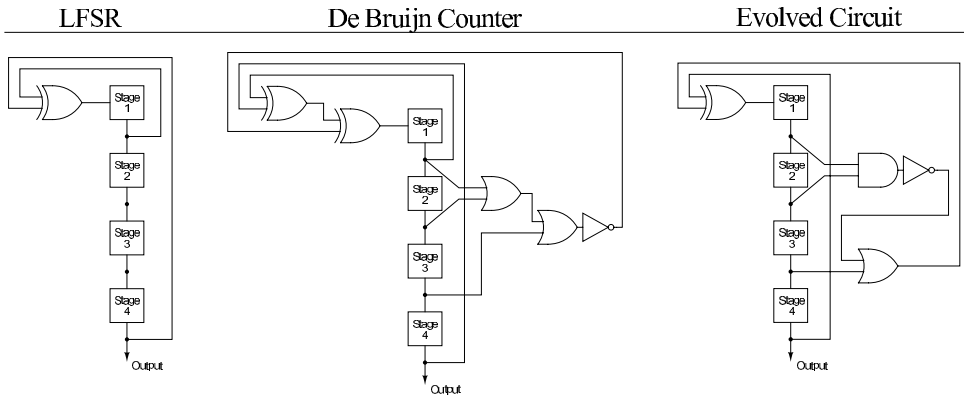

Fig. 3. Possible solutions to the problems. On the left, a Linear Feedback Shift Register (LFSR) with a period of only 15 cycles; in the center, the De Bruijn Counter that must be specifically evolved in the second task; on the right, the simplest 16 cycle circuit found.

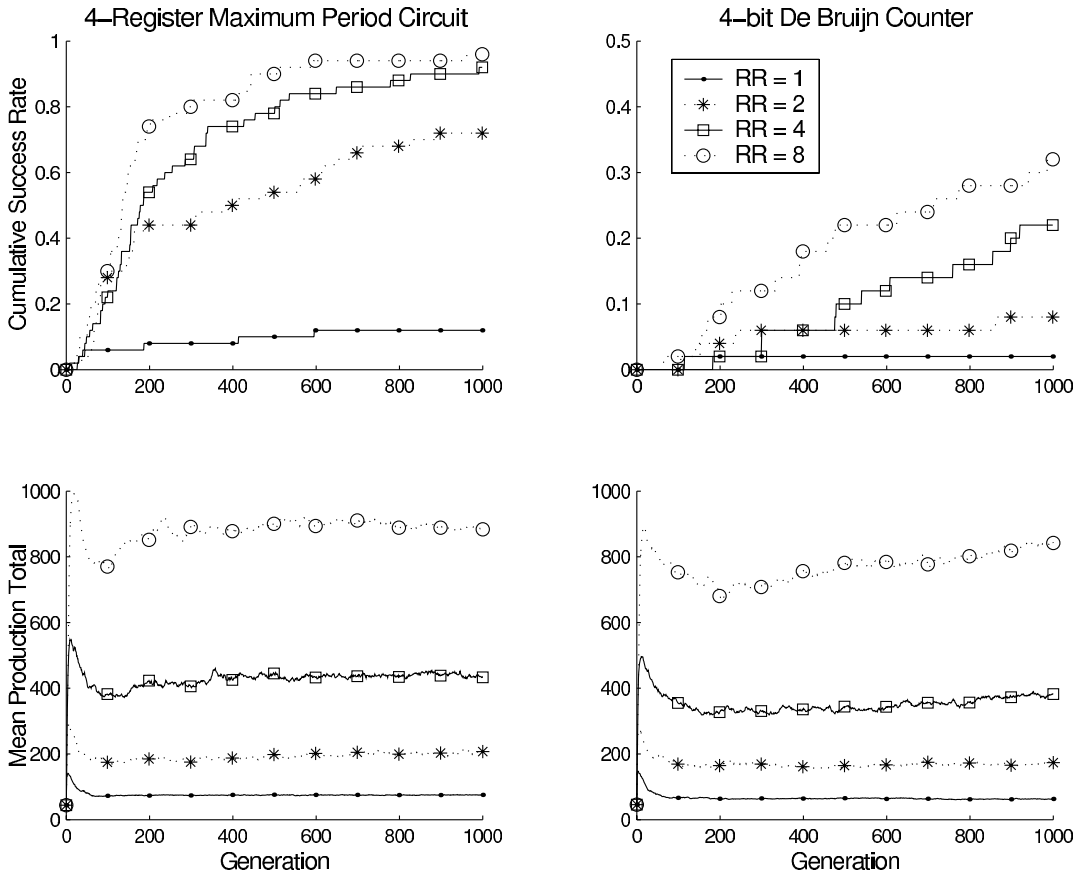

Fig. 4. Generational statistics over all runs of all configurations for population size 20: Top row shows fraction of successful outcomes; bottom row displays the mean number of productions in the shared grammar set at each generation.

confirming that it is indeed very feasible to reverse-engineer a simple PRNG through evolution.

Across these two problem tasks we observe a trend of performance improvements with increases in population size and RR. This is particularly evident for the 4-register maximum period circuit, where increases in success rate correlate with population size ($p = 0.003$ for RR = 1, Spearman rank correlation) and RR ($p = 0.017$ for size 5). The biggest improvements are observed for increases in RR, in particular between having no redundancy and having some redundancy, which is significant at $p < 0.001$ (according to a two-tailed Z-test) for all sizes except 5. Results for the De Bruijn circuit are more variable and substantial improvements are only observed for large populations, high redundancy, and, most notably, combinations thereof. For example, redundancy needs to be increased to RR = 32 to produce a significant difference ($p < 0.05$) at size 5, but only to RR = 16 at size 10, RR = 4 at size 20, and RR = 2 at size 40 and beyond. A relationship clearly exists between the two parameters, but it appears difficult to estimate an optimal RR value or population size from this.

Figures 4 and 5 illustrate how various population statistics change across generations. We note in Figure 4 (bottom) that the creation of solutions at the start (via the template grammar) leads to an early peak in the total number of productions, which quickly abates and then slowly rises again as better, larger solutions are discovered. Here, an RR value of N produces an approximately N-fold increase in the production total, although the number of productions that are part of selected solutions would not be expected to change. Yet Figure 5 (top) reveals that the presence of redundancy causes large variations in the mean size of the selected solutions. As selected means fit, we conclude that large solutions – whose creation is facilitated by redundancy – play a critical role in progressing evolution.

The drawback of exploring larger solutions is the computational expense of evaluating these. Figure 5 (bottom) demonstrates that the mean size of the evaluated solutions (i.e., the offspring) is much larger than that of selected solutions, particularly with increased RR values. Averaged

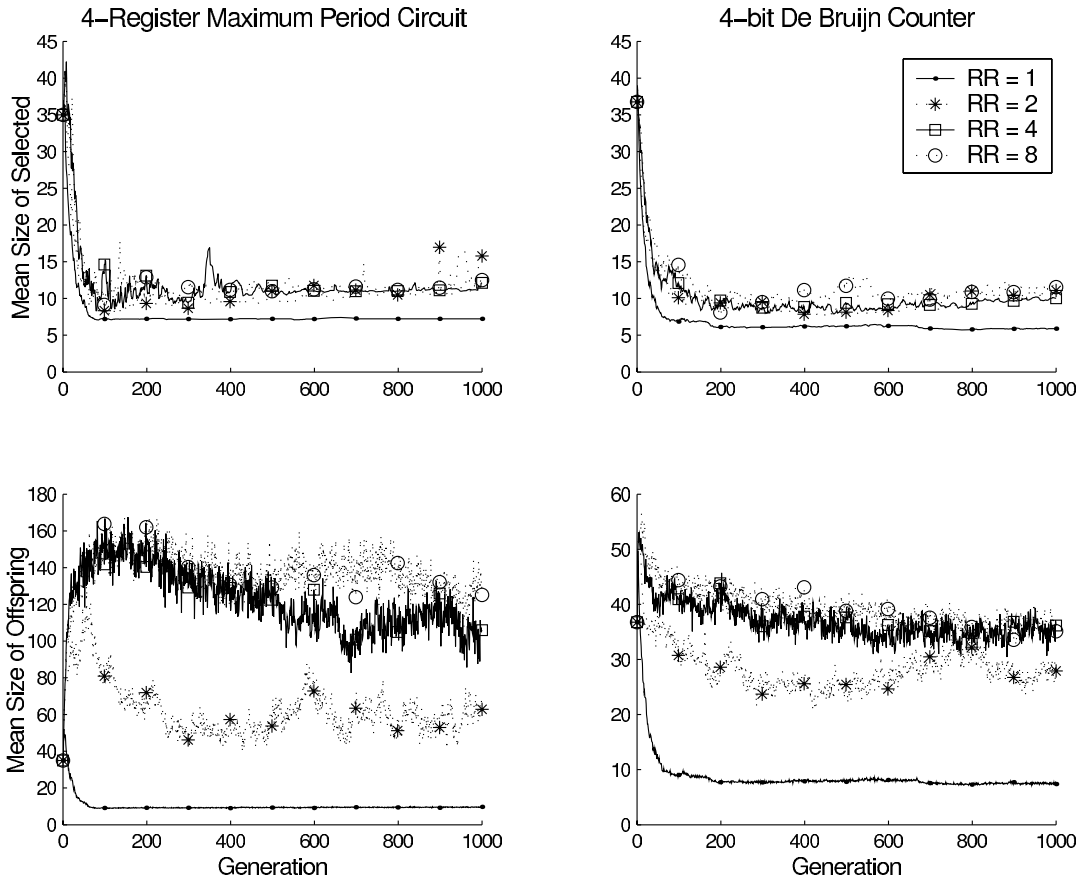

Fig. 5. Generational statistics over all runs of all configurations for population size 20: Top row shows mean size of selected solutions (those that survived beyond this generation); bottom row shows mean size of all offspring solutions.

across all generations, the size of evaluated solutions for the minimum circuit task is about $1.3\times$ the size of selected solutions for RR = 1, $5.7\times$ for RR = 2, $10.3\times$ for RR = 4, and $11.2\times$ for RR = 8. On the De Bruijn task, it is $1.3\times$ for RR = 1, $2.8\times$ for RR = 2, $3.7\times$ for RR = 4, and $3.4\times$ for RR = 8. One would expect that with redundant productions not being part of solutions, they need not be evaluated – and this remains true. In practice, the use of large, redundant building blocks in new solutions leads to a convergence slowdown comparable to that experienced with a matching increase in population size.

VII. Conclusions

Earlier studies have reported that redundancy can facilitate evolutionary search, and this study further reinforces this notion. Redundancy in SGE comprises productions that are not contributing to any existing solution candidate and whose deletion is postponed for some number of generations subsequent to their last expression. We have assessed the impact of redundant productions on the evolutionary optimization of simple circuits relating to hardware PRNG design, with more advanced research in this domain intended in future. Allowing for redundancy produces significant improvements in the performance of evolved circuits, with a significant correlation between this performance and the extent of redundancy. A significant but lesser improvement is observed when also boosting the population size.

An explanation of the success of redundancy, but also its major downside, lies in the substantial variance in the size of evaluated solutions. While redundant productions are not directly exposed to the selection objective, they are indirectly selected for their capacity at being used, i.e., being part of a solution candidate. This naturally encourages formation of large, recursive building blocks composed of many productions. Choosing one of these for a new solution candidate may invoke most or all of the other productions in this cluster, and larger clusters are more likely to have one of their productions chosen. It is a problem to which all variable-length representations are prone to a certain degree, depending on their reuse and recursion of components.

Consequently, while the production total does not rise beyond what is expected, there is a considerable increase in larger solution candidates. This appears to be in excess of what is necessary to achieve fitness, because the solutions ultimately selected are often smaller. The longer evaluation times inevitably arising from this detract from the substantial performance benefits of introducing redundancy. Redundant productions facilitate bloat, even though they are not part of any solutions - and hence not part of the bloat. Future work therefore needs to look into establishing a finer balance between encouraging redundancy in the representation yet excluding it from the derived solutions. The objective is to exploit redundancy to obtain building blocks of the right size and shape to support an effective search strategy.

REFERENCES

[1] E. De Jong and J. Pollack, "Multi-objective methods for tree size control," *Genetic Programming and Evolvable Machines*, vol. 4, no. 3, pp. 211–233, 2003.

[2] M. H. Luerssen, "Phenotype diversity objectives for graph grammar evolution," in *Advances in Natural Computation*, H. Abbass, T. Bossamaier, and J. Wiles, Eds. Singapore: World Scientific, 2005, ch. 12, pp. 159–170.

[3] M. H. Luerssen and D. M. W. Powers, "Evolving encapsulated programs as shared grammars," Submitted to *Genetic Programming and Evolvable Machines*, 2007.

[4] A. Lindenmayer, "Mathematical models for cellular interaction in development, parts I and II," *Journal of Theoretical Biology*, vol. 18, pp. 280–315, 1968.

[5] E. Boers and I. Sprinkhuizen-Kuyper, "Combined biological metaphors," in *Advances in the evolutionary synthesis of intelligent agents*, M. Patel, V. Honavar, and K. Balakrishnan, Eds. Cambridge, MA, USA: MIT Press, 2001, ch. 6, pp. 153–183.

[6] G. Hornby, "Generative representations for evolutionary design automation," Ph.D. dissertation, 2003.

[7] C. Ryan, J. Collins, and M. O'Neill, "Grammatical evolution: evolving programs for an arbitrary language," in *Proceedings of the First European Workshop on Genetic Programming*, ser. Lecture Notes in Computer Science, vol. 1391. Springer-Verlag, 1998, pp. 83–95.

[8] Y. Shan, R. McKay, R. Baxter, H. Abbass, D. Essam, and H. Nguyen, "Grammar model-based program evolution," in *Proceedings of the IEEE Congress on Evolutionary Computation*, vol. 1. IEEE Press, 2004, pp. 478–485.

[9] M. O'Neill and C. Ryan, "Grammatical evolution by grammatical evolution: The evolution of grammar and genetic code," in *Proceedings of the European Conference on Genetic Programming*, ser. Lecture Notes in Computer Science, vol. 3003. Springer-Verlag, 2004, pp. 138–149.

[10] J. Koza, *Genetic programming: on the programming of computers by means of natural selection*. Cambridge, MA, USA: The MIT Press, 1992.

[11] M. H. Luerssen and D. M. W. Powers, "Graph design by graph grammar evolution," in *Proceedings of the IEEE Congress on Evolutionary Computation*. IEEE Press, 2007.

[12] W. Langdon and R. Poli, "Fitness causes bloat," in *Second On-line World Conference on Soft Computing in Engineering Design and Manufacturing*, P. Chawdhry, R. Roy, and R. Pan, Eds. Springer-Verlag, 1997, pp. 13–22.

[13] K. Deb, S. Agrawal, A. Pratab, and T. Meyarivan, "A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii," in *Proceedings of the Parallel Problem Solving from Nature VI Conference*, ser. Lecture Notes in Computer Science, M. Schoenauer, K. Deb, G. Rudolph, X. Yao, E. Lutton, J. Merelo, and H.-P. Schwefel, Eds., vol. 1917. Springer-Verlag, 2000, pp. 849–858.

[14] T. F. Hansen, "The evolution of genetic architecture," *Annual Review of Ecology Evolution and Systematics*, vol. 37, no. 1, pp. 123–157, 2006.

[15] W. Banzhaf, "Genotype-phenotype-mapping and neutral variation – a case study in genetic programming," in *Proceedings of the Parallel Problem Solving from Nature III Conference*, ser. Lecture Notes in Computer Science, vol. 866. Springer-Verlag, 1994, pp. 322–332.

[16] I. Harvey and A. Thompson, "Through the labyrinth evolution finds a way: A silicon ridge," in *Proceedings of the First International Conference on Evolvable Systems: From Biology to Hardware*. Springer-Verlag, 1996, pp. 406–422.

[17] R. Shipman, M. Schackleton, M. Ebner, and R. Watson, "Neutral search spaces for artificial evolution: a lesson from life," in *Artificial Life: Proceedings of the Seventh International Conference on Artificial Life*. MIT Press, 2000, pp. 162–169.

[18] M. Ebner, P. Langguth, J. Albert, M. Shackleton, and R. Shipman, "On neutral networks and evolvability," in *Proceedings of the IEEE Congress on Evolutionary Computation*. IEEE Press, 2001, pp. 1–8.

[19] M. Toussaint and C. Igel, "Neutrality: A necessity for self-adaptation," in *Proceedings of the IEEE Congress on Evolutionary Computation*. IEEE Press, 2002, pp. 1354–1359.

[20] R. M. Downing, "Neutrality and gradualism: encouraging exploration and exploitation simultaneously with binary decision diagrams," in *Proceedings of the IEEE Congress on Evolutionary Computation*. IEEE Press, 2006, pp. 615–622.

[21] J. Miller and S. Smith, "Redundancy and computational efficiency in cartesian genetic programming," *IEEE Transactions on Evolutionary Computation*, vol. 10, no. 2, pp. 167–174, 2006.

[22] T. Smith, P. Husbands, and M. O'Shea, "Neutral networks in an evolutionary robotics search space," in *Proceedings of the IEEE Congress on Evolutionary Computation*. IEEE Press, 2001, pp. 136–143.

[23] J. D. Knowles and R. A. Watson, "On the utility of redundant encodings in mutation-based evolutionary search." in *Proceedings of the Parallel Problem Solving from Nature VII Conference*, ser. Lecture Notes in Computer Science, J. J. Merelo-Guervós, P. Adamidis, H.-G. Beyer, J.-L. F.-V. nas, and H.-P. Schwefel, Eds., vol. 2439. Springer-Verlag, 2002, pp. 88–98.

[24] E. Galván-López and R. Poli, "An empirical investigation of how and why neutrality affects evolutionary search," in *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM Press, 2006, pp. 1149–1156.

[25] G. Marsaglia, "The marsaglia random number cdrom including the diehard battery of tests of randomness." 1996. [Online]. Available: http://stat.fsu.edu/pub/diehard

[26] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 1, pp. 3–30, 1998.