Evaluating the Seeding Genetic Algorithm

Ben Meadows[1], Pat Riddle[1], Cameron Skinner[2], and Mike Barley[1]

[1] Department of Computer Science, University of Auckland, NZ
[2] Amazon Fulfillment Technologies, Seattle, WA

Abstract. In this paper, we present experimental results supporting early work on the Seeding Genetic Algorithm. We evaluate the algorithm's performance with various parameterisations, making comparisons to the Canonical Genetic Algorithm, and use these as guidelines as we establish reasonable parameters for the seeding algorithm. We discuss how experimental results complement and confirm aspects of the theoretical basis, such as the exclusion of the deleterious mutation operator from the new algorithm. We report on experiments on GA-difficult problems which demonstrate the Seeding Genetic Algorithm's ability to overcome local optima and systematic deception.

Keywords: genetic algorithms, evolutionary algorithms, seeding genetic algorithm

1 Introduction

Although the field of evolutionary computation has been around for over 30 years, it is still poorly understood. Evolutionary algorithms' behaviour, including the low-level interaction of their constituent parts, is often counter-intuitive. For example, Forrest and Mitchell[1] described the effects of *hitchhiking* on the genetic algorithm (GA). Hitchhiking is where suboptimal parts of an individual tend to be copied along with parts that contribute to high fitness. They also demonstrated that a naive selection scheme can lead to reduced performance by causing extinction of the fundamental *building blocks* that appear in the solution.

An analysis by Wu, Lindsay and Riolo [2] concluded that while the *crossover* operator is capable of propagating building blocks as intended, and is good at recombining building blocks, crossover's capacity for recombination is very often stunted by low building block diversity. The *mutation* operator is better at discovering new building blocks but at the cost of being more destructive to existing building blocks. While mutation is often considered to be the operator that discovers new building blocks, and investigations such as that of Wu et al. claim that mutation is better than crossover at constructing building blocks, there may be an even better mechanism for fulfilling this role. Operators that are good at discovery are also likely to be highly disruptive (for example, random search). It should be possible to explicitly treat building block discovery and combination as separate phases of the algorithm and optimise them independently. It should also be possible to adjust our selection scheme to reduce or even preclude building block extinction without unduly reducing performance.

In making these claims we build on preliminary work[3–5] on a *seeding operator* that significantly outperforms mutation on the task of building block discovery, dramatically reduces extinction events, and is able to navigate local optima. Skinner[5] mentions that it should be possible to run a more flexible 'standard' GA: one that performs well to a predictable degree without the need for fine-tuning. He wants to determine why the genetic algorithm fails in some situations where it is expected to do well, and adjust it so it does well in those situations. Exploring the degree to which he has succeeded with the *Seeding Genetic Algorithm* (SGA) will comprise a significant part of this paper.

We conduct a series of experiments to examine how the genetic algorithm behaves on a number of problems. We investigate several well-known fixed-length problems with integer fitness functions and known building block structure, on the basis that the GA's power arguably[6] comes from its ability to combine different building blocks from multiple parents. In particular we are interested in the ability of the algorithm to discover new building blocks and to select pairs of parents that have some hope of producing highly fit offspring. We also aim to establish more precisely where seeding succeeds or fails. We extrapolate from the SGA's ability to solve various parametrisations of difficult problems. The SGA adds several new parameters to the already quite complex regular genetic algorithm; our testing will therefore be quite multivariate. We aim to establish the best and most generalisable parameters for this new algorithm, and show that the goal of a more flexible GA operator has been met.

2 The Genetic Algorithm

The GA first appeared in the literature in 1975 [7]. There have been many refinements, to the extent that we suspect that the fundamental or canonical GA is seldom used in practise today other than on a few known problems with 'reasonable' parameters established by trial and error [8]. Perhaps the core problem for the GA, aside from lack of foreknowledge about the best parametrisation for a particular instance, is maintaining population diversity [9]. Numerous techniques have been demonstrated to combat early convergence, including crowding [10], clearing and speciation [11], niching and fitness sharing [12], the island model [13], and spatially distributed populations [14]. However, few of them provide a strong guarantee that they will indefinitely prevent premature convergence.

Skinner's seeding algorithm [4, 5] makes such a guarantee. We consider this work groundbreaking in that it systematically confronts questions such as

1. How does a GA actually work, rather than how should it work? and
2. Given the observed failings of the GA, how can we remodel the way it works to improve it?

3 Discovery vs Combination

We have already suggested that there is an important trade-off between discovery and combination [3–5]. As building blocks become more difficult to discover

(i.e. as the number of bits in each block increases), having a high discovery rate becomes more important. Skinner's results show that more disruptive operators are better at discovery. Unfortunately, this means that high discovery ability comes with a high probability of building block destruction. For example, uniform crossover is the best crossover operator for discovery. In the case of the mutation operator, a mutation rate of 1 (equivalent to performing random search with replacement) is best for discovery. The results imply that for the GA to perform well in the discovery phase (i.e. to make sure that building blocks that are not currently in the population are available for the next generation), random search is more effective than mutation and crossover.

Skinner's extensive analysis of the performance of the Canonical Genetic Algorithm (CGA) concludes that, in general, it is almost incapable of discovering new building blocks after just a few generations [5]. The CGA essentially has two phases: a brief, rapid combination of building blocks in the initial population, and then a long, slow discovery of the remaining building blocks, one at a time. Skinner goes on to evaluate the discovery and destruction potential of the most common genetic operators, and finds that mutation and crossover perform poorly. Mutation is only ever able to discover new blocks when there is a high prior probability that local search will be useful. In the cases tested, mutation destroys more than 99% of the blocks it discovers [5].

The problem is not lack of diversity, but that the genetic operators are simply inappropriate for the way the algorithm is meant to function[5]. However, continued building block supply is vital – especially since the GA is almost guaranteed to rapidly complete when the requisite blocks are present somewhere in the population in each generation[5]. In practise, existing blocks become extinct with disturbing frequency. Random search outperforms the common genetic operators in many circumstances [5]. What little discovery does occur in the CGA is largely due to crossover rather than mutation; mutation is largely detrimental to building block retention. It is also difficult to design a selection scheme that reliably chooses parents that are likely to have different sets of building blocks.

These results led to the Seeding Genetic Algorithm, which divides the functionality of the genetic algorithm into selection, discovery and combination. The SGA does away with mutation. It uses random search at a single initial stage coupled with ongoing sampling from a resulting 'seed pool' to perform building block discovery. It is this seeding operator which we wish to focus on.

3.1 Seeding

The design of the Seeding Genetic Algorithm is based on the observation that random search is at least as good as any other technique for discovering lowest level building blocks. The SGA draws from a seed pool of selected randomly generated individuals. This pool is generated by presampling the search space: generating a number of random individuals, measuring their fitness, and estimating the underlying average fitness of the space. Those random individuals that have fitness greater than the average are used to populate the seed pool. The

seed pool is thus initialised with a subset of (or possibly all of, if the presample size is sufficiently large) the lowest level building blocks in the problem.

When the SGA is run, candidates from the seed pool have a chance to be inserted into the main population via seeding during the crossover step. Each selected parent is replaced by a random individual drawn from the seed pool with some probability p. Seeding serves two purposes: when a particular building block has become extinct in the main population the seed pool will still retain a copy that can then be incorporated into the population via recombination; and those parts of seed pool individuals' genomes that do not contribute to fitness will be copied into the main population along with the building blocks, providing a useful diversity boost and helping prevent early convergence.

4 Problem Sets

We used several different problem sets in our analysis; we present three of them here.

The *Royal Road* problem[3] is a classic case study, designed as a GA-easy problem, but discovered to be unusually difficult [16, 17]. The Royal Road lacks deception, has a single solution, features discrete building blocks upon which an appropriate fitness function can be directly based, and can be readily solved by other optimisation techniques such as hill-climbing. A hierarchical variant [16] adds higher-order blocks, composed of lower-level ones, to the mix.

The *Hierarchical If-And-Only-If* problem[4] is used by Skinner [5] as a more "interesting" problem with multiple global optima. Described by Watson and Pollack [18], Hierarchical If-And-Only- If is a recursive problem: the bit-string is separated iteratively into halves, and rewarded with a score equal to the combined length of the two halves if all the bits in a half are homogeneous. This is a difficult problem: there are 254 blocks of six different lengths, with extremely strong interdependence. There are also two possible maximally different solutions; the fitness landscape is therefore a fractal of local optima. For each optimum, whether it is global or local, there are two less-fit optima equidistant from it. This is very difficult for a hill-climbing algorithm to solve [5].

The *Deceptive Trap* problem[5] is also designed to make optimisation algorithms such as gradient descent fail; Deceptive Trap does so at the building block level. In this deceptive problem, described by Goldberg[19], each block decreases in fitness as it comes closer to its complete form. Incremental improvements for a block lead the algorithm towards a local maxima. A complete inversion is required to go from the trapped state to the maximum for that block. Any hill-climbing type search will lead directly away from the global maximum.

[3] Our implementation of this problem, RR, following Mitchell and Forrest [15], is a 64-bit Royal Road consisting of eight adjacent, coherent blocks of eight bits each.

[4] We implement a Hierarchical If-And-Only-If problem, $HIFF$, of length 64.

[5] Our implementation, DT, is a 60-bit string consisting of 12 traps of length 5 each (following Skinner [5]).

Table 1. Mean number of generations required for the SGA to solve the RR problem with (–M) and without (+M) mutation. 2-, 5- and 32- tournament selection, rank selection, and fitness proportional selection are compared (columns) using seed probabilities of 0.1, 0.2, and 0.3 (rows). 'N/A' indicates cases where no runs found a solution.

	2-tourn.		**5-tourn.**		**32-tourn.**		**Rank**		**Fit. Pr.**	
	-M	+M	-M	+M	-M	+M	-M	+M	-M	+M
CGA		491.1		257.3		249.2		451.8		615.9
SGA										
s=0.1	203.0	N/A	98.6	100.6	117.1	94.7	177.7	314.3	417.2	523.7
s=0.2	708.0	N/A	75.5	166.9	79.2	89.3	379.6	N/A	624.4	N/A
s=0.3	N/A	N/A	193.1	472.7	76.9	96.8	N/A	N/A	N/A	N/A

5 Initial Seeding Tests

What are reasonable parameters for the SGA? In these sections, we employ the *bootstrap test* for statistical significance described in Cohen and Kim[20]. This usage is motivated by our having censored data (not all runs of the algorithm complete). The bootstrap test draws many "bootstrap samples" from the combined results of two algorithms A and B to establish whether the difference in their means is significantly different from the 0 predicted by the null hypothesis. We set $p < 0.05$ to determine statistical significance.

5.1 Seeding Probability against Mutation Rate

We first apply an experimental framework to the question of whether there is any point to retaining mutation alongside seeding; the theoretical work of Skinner determined mutation to be ineffectual as a discovery operator, but did not test this. In order to determine whether mutation is genuinely unhelpful in practice, we ran tests on RR with 2-point crossover, a presample size of 1000, and seed pool size of 50. We took the mean of 100 runs each time, testing various combinations of parameters. The results given in Table 1 are fairly typical.

The SGA (with lower seed probabilities of 0.1 to 0.2) usually performs significantly better than the CGA. Mutation typically worsens its performance, although there are some anomalies. We found that a seed probability of 0.15 was best without mutation, while a higher probability of 0.2 was better with minimal (1/1024) mutation, or a lower probability of 0.1 with normal (1/64) mutation. This suggests there is some level of interference between the mutation and seeding operators. As use of the mutation operator with the SGA is, indeed, not particularly helpful overall, we discard it for the remainder of our tests.

5.2 Seeding with Random Individuals

These results seem to bear out Skinner's [5] assertion that the usefulness of seeding lies in introducing 'superior' genetic material (building blocks). However, we wish to determine how much of the benefit of seeding comes from it preventing

Table 2. Mean number of generations to complete, standard deviation, and number of runs completing, over 1000 runs of the random-seeding SGA on the RR problem. Each column describes a different seed probability. The CGA is given as a baseline.

	CGA	**SGA**				
	(s=0)	**s=0.1**	**s=0.15**	**s=0.2**	**s=0.25**	**s=0.3**
Generations	257.3	600.0	555.7	555.9	587.5	685.0
Std. Deviation	134.5	246.1	249.7	237.5	225.9	288.7
Number completing	1000	430	606	676	398	3

early convergence by introducing 'fresh' genetic material. We separate this from building block based utility by comparing the CGA with a form of the SGA that seeds with randomly generated individuals. The test was run on RR with 5-tournament, 2-point crossover, and other standard parameters.

We are effectively comparing the power of the mutation operator in the CGA and the power of the seeding operator in the SGA to prevent early convergence.[6] The results are shown in Table 2. In every instance, either the CGA significantly outperformed the random-seeding SGA or the SGA only succeeded in a small number of runs. Since seeding with random individuals is harmful,[7] we conclude that the benefit of the seed pool comes from its introduction of building blocks.

Seeding is doing more than simply enforcing diversity; it is providing the building blocks upon which the algorithm depends. We will now investigate the various parameters of the SGA. We aim to find good seed pool sizes, seeding probabilities, and strategies for efficient and effective seed pool creation.

6 Presample Size and Seed Probability

We tested the SGA by varying the seeding probability against the mutation rate, aiming to establish a 'reasonable' seeding probability. Skinner [5] used a presample size of 1000, and chose the probability $\frac{1}{3}$ so that approximately half of the crossover operations would involve replacement of at least one parent. We conducted experiments varying the size of the presample from 500 to 750000. We tested seeding probabilities in the range 0.1 to 0.35, using typical parameters, including 5-tournament selection, 2-point crossover, and seed pool size $n = 50$.

For the remainder of this paper, we *normalise* the results of the SGA to compensate for the primary cost involved: seed pool generation.[8] We can calculate the cost of seed pool generation by assuming that the computational bottleneck

[6] The chance of the mutation operator introducing a new block is remote; mutation has an expected discovery rate of <0.001. Similarly, seeding with random individuals could technically introduce new blocks, but the chance that a random individual contains a block is 0.031, and the higher the prevalence of blocks in the main population, the less chance the offspring of a seed individual will be fit enough to survive.

[7] However, we do observe an effect whereby low levels of seeding can help by acting as a macro-mutation operator, crudely mimicking a 'normal' GA.

[8] Note that an arbitrarily large presample will otherwise always be 'best' in the sense that at some point, the presample will actually include a solution string.

Table 3. Mean generations to complete for the RR problem; comparing presample size (rows) and seed probabilities (columns). The best value is given in **bold**. Values worse than those for the CGA are given in *italics*. Numbers presented here are normalised for the cost of seed pool generation.

	0.1	**0.15**	**0.2**	**0.25**	**0.3**
500	*363.7*	*315.7*	*323.4*	*342.2*	*506.1*
1000	221.7	181.8	169.7	196.7	*394.8*
2000	126.2	99.1	91.5	107.1	235.9
5000	122.9	98.5	95.1	96.4	229.9
10000	123.3	94.9	**88.2**	101.4	199.3
20000	118.6	99.9	97.9	103.8	189.6
30000	124.6	106.6	107.4	111.4	177.3
40000	128.6	112.5	111.8	118.5	165.5
50000	135.5	122.4	119.9	125.4	167.2
100000	189.6	185.8	182.9	188.2	206.3
125000	225.2	220.4	220.8	222.9	238.6
150000	*265.2*	*258.9*	*258.4*	*260.9*	*275.6*
200000	*342.3*	*336.2*	*336.1*	*338.7*	*354.0*

is the number of fitness evaluations performed [5]. The canonical GA makes N fitness calculations per generation, depending on the population size, selection type, and number of children generated per crossover event. A presample size of N is then equivalent to running one extra generation of the CGA, and so a presample of size X has an additional cost of X/N.

6.1 Royal Road

The CGA completed RR with 2-point crossover in a mean 256.6 generations. The seeding algorithm proved able to reduce this by as much as 233 generations (90%), or by 168 generations (66%) after normalisation. Consider Table 3. The 'best' values are a presample size around 10,000 and a seed probability near 0.2. Note that the seeding algorithm still significantly outperforms the CGA with a seed probability as low as 0.1 or as high as 0.3, and continues to do so until it hits diminishing returns with a presample size $S = 150{,}000$ or higher, from which point we measured no significant improvement up to $S = 500{,}000$. The SGA performed comparably in variations of the task including use of 32-tournament selection and hierarchical versions of the royal road problem.

6.2 Hierarchical If-And-Only-If

The HIFF problem has prominent, massively interdependent building blocks, with two $(n-1)$-value fitness peaks flanking every (n)-value peak. The CGA is essentially unable to solve this problem: one in 1000 runs succeeded with 2-point crossover; the "weaker" selection operators (rank selection and 2-tournament) performed very slightly better, as they are better at escaping local optima.

The SGA was able to solve HIFF fairly easily. Early tests indicated higher seed probabilities than those for our simpler problems. However, even with a

Table 4. Tests on deceptive problems: HIFF (left) and DT (right). Each value is a mean over 100 runs to a maximum of 20,000 generations. Rows are different presample sizes; columns are different seed probabilities. The best value for each problem is given in **bold**. Numbers presented here are normalised for the cost of seed pool generation.

	0.2	**0.25**	**0.3**	**0.35**
100	8016.4	3932.5	7794.6	N/A
500	5029.7	2860.9	6794.4	N/A
1000	6521.5	2149.4	6301.2	10152.0
5000	5781.4	2261.7	4318.8	8800.1
10000	4239.2	1774.0	3087.8	9409.9
50000	3309.3	1291.0	2019.4	6071.1
100000	3505.3	1480.9	1109.4	4404.2
200000	1734.1	1126.4	1257.7	2981.6
300000	2845.7	1467.7	**1084.5**	2365.3
400000	3397.8	1394.7	1099.4	2379.8
500000	2505.6	1336.9	1101.9	2232.5
600000	2980.7	1534.6	1306.1	2074.4

	0.1	**0.15**	**0.2**
100	3182.8	4279.3	4684.1
500	1458.9	1623.0	1566.1
1000	1063.0	1251.1	966.7
5000	585.0	563.0	556.2
10000	513.6	386.3	458.0
25000	401.9	394.8	331.5
50000	372.0	**282.8**	307.4
75000	355.6	332.9	325.5
100000	371.0	360.1	356.5
150000	492.3	397.6	417.5
200000	491.4	479.5	481.1
500000	930.3	913.6	939.3

presample size of 500,000, 10% of the runs did not complete in the best case. We therefore ran a set of experiments with the maximum number of generations increased from 1000 to 20,000, using higher seed probabilities of 0.2, 0.25, 0.3, and 0.35. These tests consisted of 100 runs rather than 1000.

The CGA was run with the same parameters as a baseline; it was unable to find the solution. According to Table 4 (left), a seed probability of 0.3 and presample size around 300,000 are our best values after normalisation.[9] While a 'good' presample size is many times that for RR, even a relatively small one is clearly still useful; all runs found a solution for the three smaller seed probabilities with presamples of at least 10,000.

6.3 Deceptive Trap

DT was expected to be the hardest problem: although it has four fewer bits than the other problems, it has deception built in at the building block level (12 'trap' blocks of length 5). The CGA cannot solve DT: it did not succeed once in over 15,000 tests we ran for various parameterisations. Remarkably, the SGA solves DT even more easily than it does HIFF. With a presample size of 500,000, the SGA is sufficiently powerful to solve the DT problem 100% of the time.

However, for more reasonably-sized presamples, not all runs completed. Therefore, as with HIFF, we ran additional tests to 20,000 generations. The CGA continued to fail completely on these longer tests. In contrast, the SGA was able to find a solution 100% of the time with all tested seed probabilities with a presample of at least 25,000. Table 4 (right) suggests that the SGA had a 'spread'

[9] The seeding probability is rather sensitive in this problem: the algorithm becomes steadily more effective with higher probabilities before dropping off suddenly between 0.3 and 0.35. Note that the 0.3 value represents a close to 50% chance that at least one parent in a reproduction event will be replaced by a seed individual.

Table 5. Tests for seed pool size on the RR problem. Each value is a mean over 1000 runs to a maximum of 1000 generations. Rows are different presample sizes; columns are different seed pool sizes. The best value is given in **bold**. Values worse than those for the CGA are given in *italics*. Numbers are normalised for the cost of seed pool generation.

Presample size	5	25	50	100	500	1000
1000	*496.7*	134.8	169.7	247.3	*422.7*	*N/A*
5000	*403.5*	134.9	95.1	**78.0**	187.4	*285.6*
10000	*359.3*	124.7	88.2	83.9	113.1	191.4
50000	*329.9*	118.4	119.9	126.1	136.5	139.2
100000	*428.8*	189.8	182.9	191.9	211.6	214.6

of generally good performance for presamples ranging from tens of thousands to hundreds of thousands. Even when a seed pool of 50 was drawn from a presample of 100, the SGA was sufficiently robust to solve DT more than two-thirds of the time. This is despite the fact that such a seed pool would be filled with individuals containing blocks in various 'deceptive' states.

7 Presample Size and Seed Pool Size

To find the best seed pool size for given presample sizes, we varied the size of the seed pool from $n = 5$ to $n = 1000$. The seed probability for each experiment is fixed at its previously discovered best level. We expected to see either (a) a pattern specific to each experiment whereby good results would correlate with seed pool size as a particular fraction of the presample, or (b) a fixed 'best' seed pool size for each problem class, given some minimum presample size.[10]

We ran experiments on RR with a seed probability of 0.2. Judging by Table 5, our initial seed pool size of 50, following Skinner [5], was a good choice; 100 appears to be slightly better, but not to a statistically significant level. The viable seed pool size increases with the presample size, but the optimal size is fairly consistent. The SGA was able to improve on the performance of the CGA (256.6 generations) with as small a seed pool as 25.

Recall that the CGA was essentially unable to solve the HIFF problem. Using the previously-determined 'good' seed probability of 0.3, we achieved the results summarised in Table 6 (left). We found that the algorithm performs well with a seed pool size around 25 to 50.

We ran the same experiments on the DT problem, using the 'good' seed probability 0.175. The results are given in brief in Table 6 (right). We see a trend towards a larger seed pool size than we have encountered so far. As we increase the presample size, the first seed pool size on which all runs completed was 1000; the second was 500. The low values in the first column are a statistical artifact: fewer than 5% of runs with a seed pool of 5 solved DT, so the ones that

[10] The former is more likely *prima facie*: an arbitrarily large presample will populate the seed pool with the same number of 'bad' individuals as a small one if the same fraction is used in both cases. However, a small seed pool will also have less *diversity*.

Table 6. Tests for seed pool size on deceptive problems: HIFF (left) and DT (right). Each value is a mean over 1000 runs to a maximum of 1000 generations. Rows are different presample sizes; columns are different seed pool sizes. The best value for each problem is given in **bold** (but see text). Numbers are normalised for the cost of seed pool generation.

Presample	HIFF: varying seed pool sizes						DT: varying seed pool sizes					
size	5	25	50	100	500	1000	5	25	50	100	500	1000
1000	592.3	510.1	550.6	510.1	440.7	N/A	309.6	534.8	544.1	580.5	722.6	N/A
5000	393.6	514.5	501.7	514.5	531.5	464.0	312.4	432.4	414.5	408.3	528.5	631.4
10000	414.6	518.9	489.6	518.9	553.0	528.0	274.4	394.6	358.7	351.3	442.8	532.3
50000	399.9	**401.7**	467.3	519.2	567.9	584.6	428.8	355.6	**305.7**	**305.1**	363.2	415.0
100000	426.3	409.1	537.2	559.3	652.0	640.3	419.5	422.5	356.9	342.0	393.6	438.8

did complete likely had exceptionally high-value individuals in their seed pools. The bold values for the size 50 and 100 seed pools represent cases where 98.8 - 99.9% of runs completed, so are more legitimate 'best' choices.

8 Conclusions

We have confirmed that the Seeding Genetic Algorithm can be considerably more effective than the Canonical Genetic Algorithm, solving the main problem of the latter by maintaining a continual supply of all building blocks somewhere in the population. When combined with a suitable selection scheme this can lead to a significant reduction in number of fitness evaluations required. We have seen an 80% or greater reduction in the difficulty of problems the GA struggles with, such as the Royal Road, and overcoming problems of local optima and deception in the case of Hierarchical If-And-Only-If and Deceptive Trap, which the CGA is not equipped at all to deal with. We have found that the ideal seed pool size is not a simple fixed fraction of the presample size. It is important both to perform seeding with high-quality individuals, and to have a high enough ratio between presample and seed pool to ensure we fill the pool with useful genetic material. However, the 'best' size appears to be modulated by problem-dependent factors.

We have obtained an initial set of 'reasonable' SGA parameters; in the process, we have run more than 1,300 experiments, involving over 1,200,000 runs of the genetic algorithm. But if the SGA is to be used in a real-world domain, we cannot initialise it with the 'best' parameters, since these are almost certainly not known in advance. How close do they have to be to the best settings for the SGA to function? The majority of tested combinations actually give better results than the CGA. Default parameters of 2-point crossover, 0.9 crossover rate, 0.2 seed probability, a seed pool of 50 and a presample of 100,000 will work considerably better than the CGA on any of our problems. We do not claim that these parameters will be universally appropriate: they may not necessarily carry over to wider domains or more complex building block situations, but we are hopeful. We have added several more parameters to the already change-sensitive genetic algorithm; the seeding probability in particular may require fine-tuning on a case-by-case basis. In some domains it may be appropriate to run "exploratory tests" to narrow the parameters down.

Many extensions to the 'canonical' GA have addressed the failings we overcame here. The utility of the seeding operator lies in its ability to prevent early convergence *and* overcome local optima *and* solve deceptive problems. Note that, the fraction of individuals in the next generation with some new (high quality) genetic material is equal to those cases where both parents are seeded, plus those where one parent is seeded and crossover occurs, plus half those where one parent is seeded and crossover does not occur. This is a fixed rate, the same from generation 1 to generation 1000. This gives the SGA an advantage over other GA extensions (such as geographically distributed populations or the island model), which delay early convergence rather than guaranteeing its prevention.

9 Future Research

Work on the seeding algorithm has been quite preliminary, but has raised many avenues of inquiry. The relationship between selection pressure and the best seed probability needs explored further. The design of our selection operator could be refined (i.e., possibly made more aggressive) taking advantage of the SGA's strengths. Finally, these results, although promising, only use a naive implementation of seeding. Exploring more sophisticated mechanisms will likely be of considerable interest. The replacement strategy (random replacement), too, is naive. There appears to be a correlation between selection pressure and the best seed replacement probability, and this warrants further investigation.[11]

The SGA could be refined. It may be possible to infer the relative merits of seeding at each point in the algorithm, allowing us to dynamically 'throttle' the seeding probability to an appropriate level. We could experiment with creating the first generation of the primary population entirely from seed pool individuals, or injecting seed individuals into the population only when the fitness of the offspring or parents in some crossover event fall below some threshold. Different policies for generating the seed pool should also be explored. For example, choosing those presample individuals that have a fitness value that is more than one standard deviation above the underlying average fitness. Finally, the way to truly prove the worth of the SGA is to run it on more complex problems. If the algorithm succeeds on real-world domains, we can say without reservation that it has succeeded as an innovation in the field of evolutionary algorithms.

References

1. Forrest, S., Mitchell, M.: Relative building-block fitness and the building-block hypothesis. In Whitley, L., ed.: Foundations of genetic algorithms. (1993) 109–126

[11] At the beginning of a GA run there are many building blocks that have not been combined into one individual, whereas later in the run most individuals contain similar blocks, so a replacement scheme that varies the probability throughout the run is likely to work better than a fixed probability. Even better would be to examine the set of possible parents and detect when a given pair of parents is likely or unlikely to contain different blocks (e.g., use domain knowledge to detect siblings or cousins).

2. Wu, A.S., Lindsay, R.K., Riolo, R.L.: Empirical observations on the roles of crossover and mutation. In Bäck, T., ed.: Proc. 7th International Conference on Genetic Algorithms. (1997) 362–369
3. Skinner, C., Riddle, P.: Expected rates of building block discovery, retention and combination under 1-point and uniform crossover. In: Parallel Problem Solving from Nature-PPSN VIII, Springer (2004) 121–130
4. Skinner, C., Riddle, P.: Random search can outperform mutation. In: IEEE Congress on Evolutionary Computation (CEC 2007). (2007)
5. Skinner, C.: On the discovery, selection and combination of building blocks in evolutionary algorithms. PhD thesis, Citeseer (2009)
6. Jansen, T.: Real royal road functions—where crossover provably is essential. In: GECCO: Proceedings of the Genetic and Evolutionary Computation Conference, Morgan Kaufmann Pub (2001) 375–382
7. Holland, J.: Adaptation in artificial and natural systems. Ann Arbor: The University of Michigan Press (1975)
8. Srinivas, M., Patnaik, L.M.: Adaptive probabilities of crossover and mutation in genetic algorithms. IEEE Transactions on Systems, Man and Cybernetics **24**(4) (1994) 656–667
9. Luke, S., Balan, G.C., Panait, L., Cioffi-Revilla, C., Paus, S.: Mason: A java multi-agent simulation library. In: Proceedings of Agent 2003 Conference on Challenges in Social Simulation. Volume 9. (2003)
10. De Jong, K.A.: Analysis of the behavior of a class of genetic adaptive systems. PhD thesis, University of Michigan Ann Arbor, MI, (1975)
11. Pétrowski, A.: A clearing procedure as a niching method for genetic algorithms. In: Proceedings of IEEE International Conference on Evolutionary Computation. (1996) 798–803
12. Darwen, P., Yao, X.: Every niching method has its niche: Fitness sharing and implicit sharing compared. In: Parallel Problem Solving from Nature—PPSN IV. Springer (1996) 398–407
13. Whitley, D., Rana, S., Heckendorn, R.B.: The island model genetic algorithm: On separability, population size and convergence. Journal of Computing and Information Technology **7** (1999) 33–48
14. Sarma, J., De Jong, K.: An analysis of local selection algorithms in a spatially structured evolutionary algorithm. In: Proceedings of the Seventh International Conference on Genetic Algorithms. (1997) 181–186
15. Mitchell, M., Forrest, S.: B. 2.7. 5: Fitness landscapes: Royal road functions. Handbook of evolutionary computation (1997)
16. Mitchell, M., Forrest, S., Holland, J.H.: The royal road for genetic algorithms: Fitness landscapes and ga performance. In: Proceedings of the first european conference on artificial life, Cambridge: The MIT Press (1992) 245–254
17. Mitchell, M., Holland, J.H., Forrest, S.: When will a genetic algorithm outperform hill climbing? Advances in neural information processing systems (1994) 51–51
18. Watson, R.A., Pollack, J.B.: Recombination without respect: Schema combination and disruption in genetic algorithm crossover. In: Proceedings of Genetic and Evolutionary Computation Conference (GECCO). (2000) 112–119
19. Goldberg, D.E.: Simple genetic algorithms and the minimal, deceptive problem. Genetic algorithms and simulated annealing **74** (1987)
20. Cohen, P., Kim, J.: A bootstrap test for comparing performance of programs when data are censored, and comparisons to Etzioni's test. Technical report, University of Massachusetts (1993)